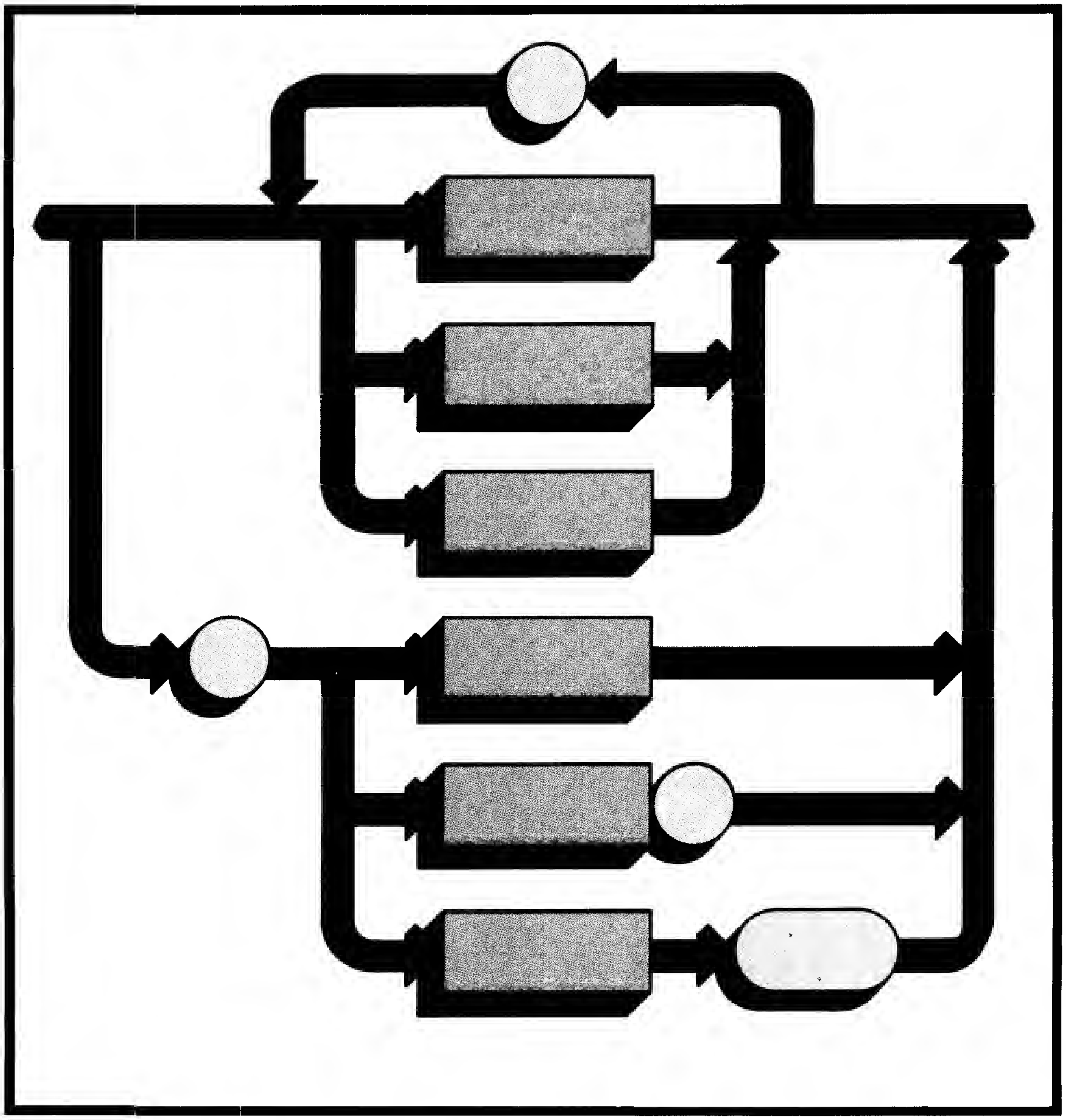
HP 9000 Series 200 Computers

HEWLETT PACKARD

HP Pascal Language Reference



HP Pascal Language Reference
for the HP 9000 Series 200
Computers

Manual Part No. 98615-90050

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

February 1984...First Edition

Table of Contents

Keyword Dictionary

Introduction

Niklaus Wirth designed the programming language Pascal in 1968 as a vehicle for teaching the fundamentals of structured programming and as a demonstration that it was possible to efficiently and reliably implement a "non-trivial" high level language. Since then, Pascal has established itself as the dominant programming language in university-level computer science courses. It has also become an important language in commercial software projects, especially in systems programming.

Hewlett-Packard Standard Pascal (HP Pascal) is a company-standard language currently implemented on several Hewlett-Packard computers and is a superset of American National Standards Institute (ANSI) Pascal.

This section outlines the organization of this manual and summarizes the differences between Pascal and HP Pascal. The experienced Pascal programmer may use these summaries as a guide for further study of unfamiliar features.

Manual Organization

This manual is a Language Reference for HP Pascal. Here you will find a description for each keyword (reserved words and standard identifiers) recognized by HP Pascal. In addition to the keywords, this manual contains entries for topics important to HP Pascal but not necessarily related to a particular keyword.

After the keyword section, you will find "implementation" sections. These sections describes HP Pascal for your particular computer. This information includes the minimum and maximum ranges for numeric values, restrictions on the sizes of variables, compiler options, system programming extensions, and error codes.

Notation

Throughout this document, HP Pascal reserved words and directives appear in uppercase letters, e.g. BEGIN, REPEAT, FORWARD. Standard identifiers appear in lowercase letters, in a typewriter-like type-style, e.g. `readln`, `maxint`, `text`. General information concerning an area of programming (a topic) appears as an entry with initial capitalization, e.g. Scope, Comments, Standard Procedures and Functions.

Where to Start

If you are totally unfamiliar with the Pascal programming language, this manual is not the place to start learning. Like a dictionary, a reference contains the facts, but trying to learn a language by reading its dictionary is a very difficult task. There are many introductory texts available that make learning Pascal much more enjoyable.

If no other book is currently available, do not try to read this manual from cover to cover. Start, instead, by reading the topics covered in this manual. Here is a partial list to get you started.

- Symbols, Identifiers, and Reserved Words
- Operators, Numbers, and Expressions
- Constants, Types, and Variables
- Statements, Assignment, Procedures, and Functions
- Programs and Modules

When you have read all of the topics and studied the keywords, you may be able to write a working program. Be sure to also read the implementation section of this manual. There are several examples of working programs throughout this manual. However, there are more "partial" examples which only show the area of interest for a particular keyword.

If you are familiar with Pascal but not HP Pascal, you may only need to refer to the implementation section of this manual. However, HP Pascal has features not found in other implementations. See the next section and the topics describing strings and modules.

If you are familiar with HP Pascal, start reading the implementation section at the back of this manual. The keyword section may prove handy when you want to check the syntax or semantics of a particular keyword.

HP Standard Pascal

The following is a list of the HP Pascal features which are extensions of ANSI Standard Pascal. For the full description of a feature, refer to the appropriate keyword or topic.

Originally, the term "string" referred to any PACKED ARRAY OF `char` with a starting index of 1. HP Pascal, however, supports the standard type `string`. To avoid confusion, the term PAC is used for the type PACKED ARRAY OF `char`.

Assignment Compatibility

If T1 is a PAC variable and T2 is a string literal (or PAC variable), then T2 is assignment compatible with T1 provided that T2 is not longer than T1. If T2 is shorter than T1, the system will pad T1 with blanks.

If T1 is `real` and T2 is `longreal`, the system truncates T2 to `real` before assignment.

CASE Statement

The reserved word OTHERWISE may precede a list of statements and the reserved word END in a CASE statement. If the case selector evaluates to a value not specified in the case constant list, the system executes the statements between OTHERWISE and END (see CASE). Also, subranges may appear as case constants.

Compiler Options (Directives)

Compiler options appear between dollar signs ($). HP Pascal has five options: ANSI, PARTIAL_EVAL, LIST, PAGE, and INCLUDE. The ANSI option sets the compiler to identify in the listing when source code includes features which are not legal in ANSI Standard Pascal. PARTIAL_EVAL permits the partial evaluation of boolean expressions. LIST allows the suppression of the compiler listing. PAGE causes the listing to resume on the top of the next page. INCLUDE specifies a source file which the compiler will process at the current position in the program.

Other options are implementation defined. See the implementation section of this manual for complete details.

Constant Expressions

The value of a declared constant may be specified with a constant expression. A constant expression returns an ordinal value and may contain only declared constants, literals, calls to the functions `ord`, `chr`, `pred`, `succ`, `hex`, `octal`, `binary`, and the operators `+`, `-`, `*`, `DIV`, and `MOD`.

A constant expression may appear anywhere that a constant may appear.

Constructors (Structured Constants)

The value of a declared constant can be specified with a constructor. In general, a constructor establishes values for the components of a previously declared array, record, string or set type. Record, array, and string constructors may only appear in a CONST section of a declaration part of a block. Set constructors, on the other hand, may also appear in expressions in executable statements and their typing is optional.

Declaration Part

In the declaration part of a block, you can repeat and intermix the CONST, TYPE, and VAR sections.

Halt Procedure

The `halt` procedure causes an abnormal termination of a program.

Heap Procedures

The procedure `mark` marks the state of the heap. The procedure `release` restores the state of the heap to a state previously marked. This has the effect of deallocating all storage allocated by the `new` procedure since the program called a particular `mark`.

Identifiers

The underscore character (_) may appear in identifiers, but not as the first character.

File I/O

A file may be opened for direct access with the procedure `open`. Direct access files have a maximum number of components, indicated by the function `maxpos`, and the current number of written components, indicated by the function `lastpos`. The procedure `seek` places the current position of a direct access file at a specified component. Data can be read from a direct access file or write to it with the procedures `readdir` or `writedir`, which are combinations of `seek` and the standard procedures `read` or `write`. A textfile cannot be used as a direct access file.

A file may be opened in the "write-only" state without altering its contents using the procedure `append`. The current position is set to the end of the file.

Any file may be explicitly closed with the procedure `close`.

To permit interactive input, the system defines the primitive file operation `get` as "deferred get".

The procedure `read` accepts any simple type as input. Thus, it is possible to read a `boolean` or enumerated value from a file. It is also possible to read a value which is a packed array of `char` or `string`.

The procedure `write` accepts identifiers of an enumerated type as parameters. An enumerated constant may be written directly to a file.

The function `position` returns the index of the current position for any file which is not a textfile. The function `linepos` returns the integer number of characters which the program has read from or written to a textfile since the last line marker.

The procedures `page`, `overprint`, and `prompt` operate on textfiles. `Page` causes a page eject when a text file is printed. `Overprint` causes the printer to perform a carriage return without a line feed, effectively overprinting a line. `Prompt` flushes the output buffer without writing a line marker. This allows the cursor to remain on the same screen line when output is directed to a terminal.

Function Return

A function may return a structured type, except the type file. That is, a function may return an array, record, set or string.

Longreal Numbers

The type `longreal` is identical with the type `real` except that it provides greater precision. The letter "L" precedes the scale factor in a longreal literal.

Minint

The standard constant `minint` is defined in the HP Pascal. The value is implementation dependent.

Record Variant Declaration

The variant part of a record field list may have a subrange as a case constant.

String Literals

HP Pascal permits the encoding of control characters or any other single ASCII character after the sharp symbol (#). For example, the string literal #G represents CTRL-G (i.e. the bell). A character may also be encoded by specifying its value (0..255) after the sharp symbol. For example, #7 represents CTRL-G.

String Type

HP Pascal supports the predefined type `string`. A `string` type is a packed array of `char` with a declared maximum length and an actual length that may vary at run time.

A variable of type `string` may be compared with a similar variable or a string literal, or assign a string or string literal to a string.

Several standard procedures and functions manipulate strings.

- `Strlen` returns the current length of a string;
- `Strmax` the maximum length.
- `Strwrite` writes one or more values to a string;
- `Strread` reads values from a string.
- `Strpos` returns the position of the first occurrence of a specified string within another string.
- `Strltrim` and `strrtrim` trim leading and trailing blanks, respectively, from a string.
- `Strrpt` returns a string composed of a designated string repeated a specified number of times.
- `Strappend` appends one string to another.
- `Str` returns a specified portion of a string, i.e. a substring.
- `Setstrlen` sets the current length of a string without changing its contents.
- `Strmove` copies a substring from a source string to a destination string.
- `Strinsert` inserts one string into another.
- `Strdelete` deletes a specified number of characters from a string.

WITH Statement

The record list in a WITH statement may include a call to a function which returns a record as its result (see WITH).

Numeric Conversion Functions

The functions `binary`, `octal`, and `hex` convert a parameter of type `string` or PAC, or a string literal, to an integer. `Binary` interprets the parameter as a binary value; `octal` as an octal value; `hex` as a hexadecimal value.

Modules

HP Pascal supports separately compiled program fragments called modules. Modules may be used to satisfy the unresolved references of another program or module.

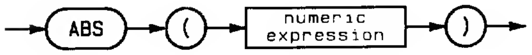
Typically, a module "exports" types, constants, variables, procedures, and functions. A program can then "import" a module to satisfy its own references.

This mechanism allows commonly used procedures and functions to be compiled separately and used by more than one program without having to include them in each program.

See MODULE.

abs

This function computes the absolute value of its argument.



Semantics

The function `abs(x)` computes the absolute value of the numeric expression x. If x is an integer value, the result will also be an integer.

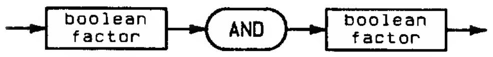
A error may result from taking the absolute value of `minint`.

Examples

Input	Result
`abs(-13)`	`13 {integer result}`
`abs(-7.11)`	`7.110000E+00`

AND

This boolean operator returns `true` or `false` based on the logical `AND` of the boolean factors.



Semantics

The logical `AND` is shown in this table.

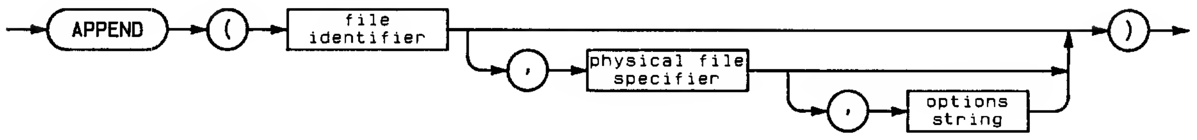
X	Y	X AND Y
false	false	false
false	true	false
true	false	false
true	true	true

Example Code

```
VAR
   bit6, bit7 : boolean;
   counter    : integer;

BEGIN
  ...
  IF bit6 AND bit7 THEN counter := 0;
  ...
  IF bit6 AND (counter = 0) THEN bit7 := true;
END
```

append

This procedure allows data to be added to an existing file.



Item	Description/Default	Range Restrictions
file identifier	name of a logical file	file cannot be of type text
physical file specifier	name to be associated with f; must be a string expression or PAC variable	-
options string	a string expression or PAC variable	implementation dependent

Examples

```
append(file_var)
append(file_var,phy_file_spec)
append(file_var,phy_file_spec,opt_str)
append(fvar,'SHORTFILE')
```

Semantics

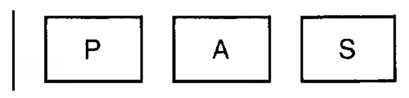
The procedure `append`(f) opens file f in the write-only state and places the current position immediately after the last component. All previous contents of f remain unchanged. The `eof`(f) function returns true and the file buffer f^ is undefined. Data may now be written on f.

If f is already open, `append` closes and then reopens it. If a file name is specified, the system closes any physical file previously associated with f.

Illustration

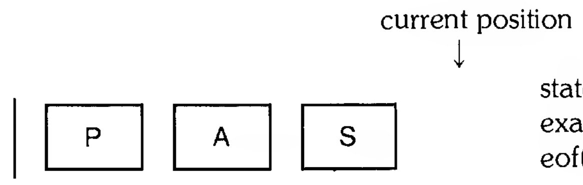
Suppose examp_file is a closed file of `char` containing three components. In order to open it and write additional material without disturbing its contents, we call append.

{initial condition}



state: closed

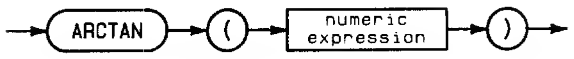
append(examp_file);



state: write-only
examp_file^: undefined
eof(examp_file): true

arctan

This function returns the principal value of the angle which has the tangent equal to the argument. This is the arctangent function.



Examples

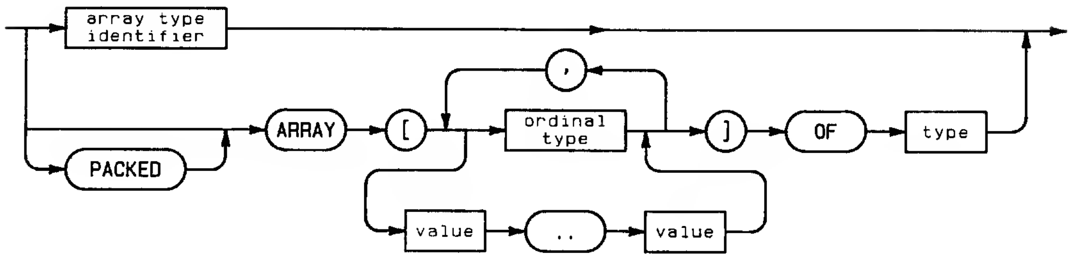
Input	Result
`arctan(num_exp)`	
`arctan(2)`	`1.107149E+00`
`arctan(-4.002)`	`-1.32594E+00`

Semantics

The result is in radians within the range $-\pi/2..\pi/2$. This function returns a real for integer or real arguments, and longreal for longreal arguments.

ARRAY

An array is a fixed number of components which are all of the same type.



Semantics

Array Declarations

An array type definition consists of the reserved word ARRAY, an index type in square brackets, the reserved word OF, and the component type. The reserved word PACKED may precede ARRAY. It instructs the compiler to optimize storage space for the array components.

A computable index designates each component of an array.

The index type must be an ordinal type. The component type may be any simple, structured, or pointer type, including a file type. The symbols (. and .) may replace the left and right square brackets, respectively.

An array type is a user-defined structured type.

A component of an array may be accessed using the index of the component in a selector.

In ANSI Standard Pascal, the term "string" designates a packed array of `char` with a starting index of 1. HP Pascal defines a standard type `string` which is identical with a packed array of `char` except that its actual length may vary at run time. To distinguish these two data types, the acronym PAC will denote

```
PACKED ARRAY [1..n] OF char;
```

throughout this manual.

The maximum number of elements is implementation defined.

Permissible Operators

assignment:	`:=`
relational (string or PAC):	`<, <=, =, <>, >=, >`

Standard Procedures

array parameters: `pack, unpack`

Example Code

```
TYPE
    name    = PACKED ARRAY [1..30] OF char; {PAC type}
  list    = ARRAY [1..100] OF integer;
  strange = ARRAY [boolean] OF char;
  flag    = ARRAY [(red, white, blue)] OF 1..50;
  files   = ARRAY [1..10] OF text;
```

Multi-Dimensioned Arrays

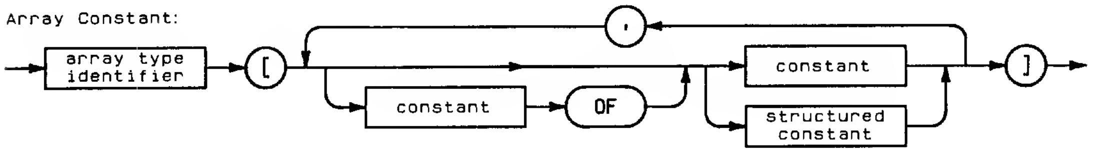
If an array definition specifies more than one index type or if the components of an array are themselves arrays, then the array is said to be multi-dimensioned. The maximum number of array dimensions is implementation defined.

```
TYPE
  { equivalent definitions of truth }
  truth  = ARRAY [1..20] OF
              ARRAY [1..5] OF
                 ARRAY [1..10] OF boolean;
  truth  = ARRAY [1..20] OF
              ARRAY [1..5, 1..10] OF boolean;
  truth  = ARRAY [1..20, 1..5] OF
              ARRAY [1..10] OF boolean;
  truth  = ARRAY [1..20, 1..5, 1..10] OF boolean;
```

Array Constants and Array Constructors

An array constant is a declared constant defined with an array constructor which specifies values for the components of an array type.

An array constructor consists of a previously defined array type identifier and a list of values in square brackets. Each component of the array type must receive a value which is assignment compatible with the component type.



Within the square brackets, the reserved word OF indicates that a value occurs repeatedly. For example, 3 OF 5 assigns the integer value 5 to three successive array components. The symbols (. and .) may replace the left and right square brackets, respectively. An array constant may not contain files.

Array constructors are only legal in a CONST section of a declaration part. They cannot appear in other sections or in executable statements.

An array constant may be used to initialize a variable in the executable part of a block. You may also access individual components of an array constant in the body of a block, but not in the definition of other constants (see Array Selector).

Values for all elements of the structured type must be specified and must have a type identical to the type of the corresponding elements.

Example Code

```
TYPE
  boolean_table = ARRAY [1..5] OF boolean;
  table         = ARRAY [1..100] OF integer;
  row           = ARRAY [1..5] OF integer;
  matrix        = ARRAY [1..5] OF row;
  color         = (red, yellow, blue);
  color_string  = PACKED ARRAY [1..6] OF char;
  color_array   = ARRAY [color] OF color_string;

CONST
  true_values   = boolean_table [5 OF true];
  init_values1  = table [100 OF 0];
  init_values2  = table [60 OF 0, 40 OF 1];
  identity      = matrix [row [1, 0, 0, 0, 0],
                          row [0, 1, 0, 0, 0],
                          row [0, 0, 1, 0, 0],
                          row [0, 0, 0, 1, 0],
                           row [0, 0, 0, 0, 1]];
  colors        = color_array [color_string ['RED', 3 OF ' '],
                               color_string ['YELLOW'],
                               color_string ['BLUE', 2 OF ' ']];
```

In the last example, the type of the array component is `char`, yet both string literals and characters appear in the constructor. This is one case where a value (string literal) is assignment compatible with the component type (`char`). Alternatively, you could write

```
colors = color_array['RED','YELLOW','BLUE'];
```

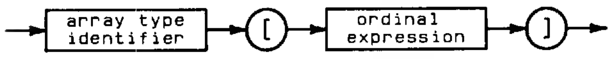
for the last constant definition.

The name of the previously declared literal string constant may be specified within a structure constant.

```
CONST
   red    = 'red   ';
   yellow = 'yellow';
   blue   = 'blue  ';

   colors  =  color_array [ color_string[red];
                            color_string[yellow];
                            color_string[blue] ];
```

Array Selector

An array selector accesses a component of an array. The selector follows an array designator and consists of an ordinal expression in square brackets.



The expression must be assignment compatible with the index type of the array. An array designator can be the name of an array, the selected component of a structure which is an array, or a function call which returns an array. The symbols (. and .) may replace the left and right brackets, respectively. The component of a multiply-dimensioned array may be selected in different ways (see example).

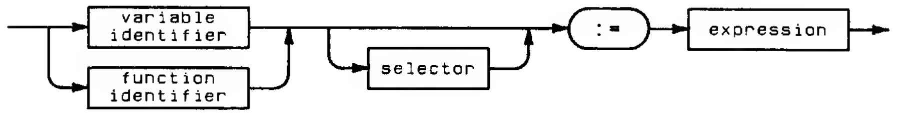
For a string or PAC type, an array selector accesses a single component of a string variable, i.e. a character.

Example Code

```
PROGRAM show_arrayselector;
TYPE
  a_type = ARRAY [1..10] OF integer;
VAR
  m,n          : integer;
  simp_array   : ARRAY [1..3] OF 1..100;
  multi-array  : ARRAY [1..5,1..10] OF integer;
  p            : ^a_type;
 BEGIN
   .
  m:= simp_array[2];        {Assigns current value of 2nd      }
   .                        {component of simp_array to m.     }
  multi_array[2,9]:= m;     {These are                         }
  multi_array[2][9]:= m;    {equivalent.                       }
   .
  n:= p^[m MOD 10 + 1] * m  {Dynamic array with computed       }
END.                        { selector.                        }
```

Assignment

An assignment statement assigns a value to a variable or a function result. The assignment statement consists of a variable or function identifier, an optional selector, a special symbol (: =), and an expression which computes a value.



The receiving element may be of any type except file, or a structured type containing a file type component. An appropriate selector permits assignment to a component of a structured variable or structured function result.

The type of the expression must be assignment compatible with the type of the receiving element (see below).

Types must be identical except when an implicit conversion is done, or a run-time check is performed which verifies that the value of the expression is assignable to the variable.

Example Code

```
FUNCTION show_assign: integer;

  TYPE
    rec = RECORD
            f: integer;
            g: real;
          END;

    index = 1..3;
    table = ARRAY [index] OF integer;

  CONST
    ct = table [10, 20, 30];
     cr = rec [f:2, g:3.0];

  VAR
    s: integer;
    a: table;
    i: index;
    r: rec;
    p1,
    p: ^integer;
    str: string[10];

  FUNCTION show_structured: rec;
    BEGIN                         {Assign to a          }
       show_structured.f := 20;   {part of the record,  }
       show_structured := cr;     {whole record,        }
       show_assign := 50;         {outer function.      }
     END;
```

```
BEGIN {show_assign}          {Assign to a                     }
  s := 5; i:= 3;             {simple variable,                }
  a := ct;                   {array variable,                 }
  a [i] := s + 5;            {subscripted array variable,     }
  r := cr;                   {record variable,                }
  r.f := 5;                  {selected record variable,       }
  p := p1;                   {pointer variable,               }
  p^ := r.f - a [i];         {dynamic variable,               }
  str := 'Hi!';              {string variable,                }
  show_assign := p^;         {function result variable,       }
END;  {show_assign}
```

Assignment Compatibility

A value of type T2 may only be assigned to a variable or function result of type T1 if T2 is assignment compatible with T1. For T2 to be assignment compatible with T1, any of the following conditions must be true:

1. T1 and T2 are type compatible types which are neither files nor structures that contain files.
2. T1 is `real` or `longreal` and T2 is `integer` or an integer subrange. The compiler converts T2 to `real` or `longreal` prior to assignment.
3. T1 is `longreal` and T2 is `real`. The compiler converts T2 to `longreal` prior to assignment.
4. T1 is `real` and T2 is `longreal`. The compiler rounds T2 to the precision of T1 prior to assignment.

Furthermore, a run-time or compile-time error will occur if the following restrictions are not observed:

If T1 and T2 are type compatible ordinal types, the value of type T2 must be in the closed interval specified by T1.

If T1 and T2 are type compatible set types, all the members of the value of type T2 must be in the closed interval specified by the base type of T1.

A special set of restrictions applies to assignment of string literals or variables of type `string`, PAC, or `char` (see below).

Special Cases

The pointer constant NIL is both type compatible and assignment compatible with any pointer type.

The empty set [] is both type compatible and assignment compatible with any set type.

String Assignment Compatibility

Certain restrictions apply to the assignment of string literals or variables of the type `string`, packed array of `char` (PAC), or `char`.

1. If T1 is a string variable, T2 must be a string variable or a string literal whose length is equal to or less than the maximum length of T1. T2 cannot be a PAC or char variable. Assignment sets the current length of T1.
2. If T1 is a PAC variable, T2 must be a PAC or a string literal whose length is less than or equal to the length of T1. T1 will be blank filled if T2 is a string literal or PAC which is shorter than T1. T2 cannot be a string or a char variable. (See table below.)
3. If T1 is a char variable, T2 may be a char variable or a string literal with a single character. T2 cannot be a string or PAC variable.

The following table summarizes these rules. The standard function `strmax`(s) returns the maximum length of the string s. The standard function `strlen`(s) returns the current length of the string s.

String constants are considered string literals when they appear on the right side of an assignment statement.

Any string operation on two string literals, such as the concatenation of two string literals, results in a string of string type.

String, PAC, and String literal Assignment

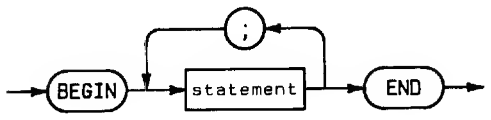
T1: = T2	string	PAC	char	String Literal
string	Only if strmax(T1) >= strlen(T2)	Not allowed	Not allowed	Only if strmax(T1) >= strlen(T2)
PAC	Not allowed	Only if T1 length >= T2 length T2 is padded if necessary	Not allowed	Only if T1 length >= strlen(T2) T2 is padded if necessary
char	Not allowed	Not allowed	Yes	Only if strlen(T2) = 1

Note

The `strlen` function can only be used with strings, not PAC's.

BEGIN

This reserved word indicates the beginning of a compound statement or block.



Semantics

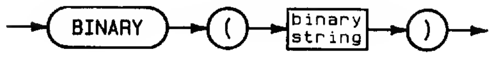
BEGIN indicates to the compiler that a compound statement or block follows.

Example Code

```
PROGRAM show_begin(input, output);

  VAR
    running : boolean;
    i, j    : integer;

   BEGIN
    i := 0;
     j := 1;
    running := true;
    writeln('See Dick run.');
    writeln('Run Dick run.');
    IF running then
       BEGIN
         I := i + 1;
         J := j - 1;
       END;
  END;
END.
```

binary

This function converts a binary string expression or PAC into an integer.



Item	Description/Default	Range Restrictions
binary string	string expression or PAC variable	implementation dependent

Examples

```
Input                                          Result
binary(strng)
binary('10011')                                  19
-binary('10011')                                -19
```

If your particular implementation used 32-bit 2's complement notation, the following example would also work.

```
binary('11111111111111111111111111101101')      -19
```

Semantics

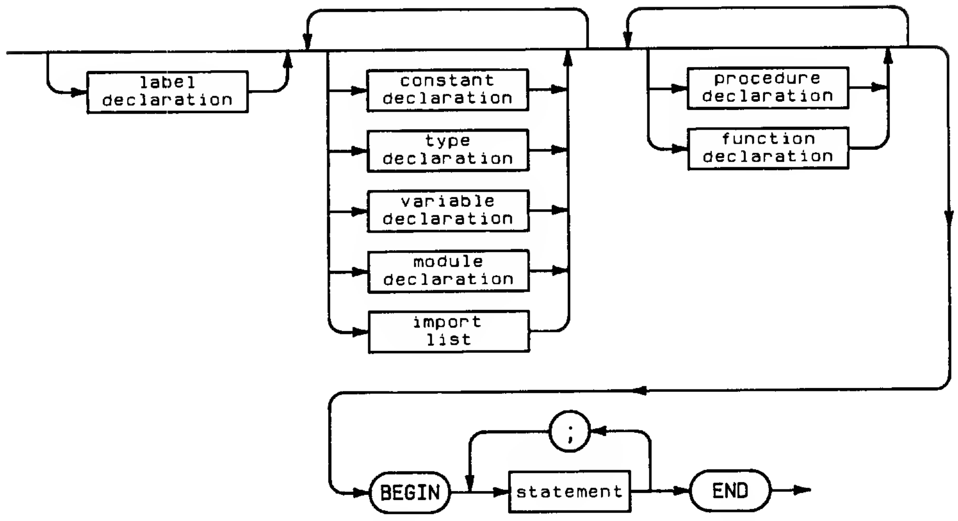
The string or PAC is interpreted as a binary value.

The three numeric conversion functions are `binary`, `hex`, and `octal`. All three accept arguments which are string or PAC variables, or string literals. The compiler ignores leading and trailing blanks in the argument. All other characters must be legal digits in the indicated base.

Since `binary`, `hex`, and `octal` return an integer value, all bits must be specified if a negative result is desired. Alternatively, you may negate the positive representation.

Blocks

A block is syntactically complete section of code.



Semantics

There are two parts to a block, the declaration part and the executable part. Blocks may be nested. All objects appearing in the executable part must be defined in the declaration part or in the declaration part of an outer block.

Note

MODULE declarations and IMPORT lists can not appear in inner blocks. (i.e. in procedures or functions)

boolean

This predefined ordinal type indicates logical data.

Example

```
VAR
  loves_me: boolean;
```

HP Pascal predefines the type `boolean` as:

```
TYPE
  boolean = (false, true);
```

The identifiers `false` and `true` are standard identifiers, where `true > false`.

`Boolean` is a standard simple ordinal type.

Permissible Operators

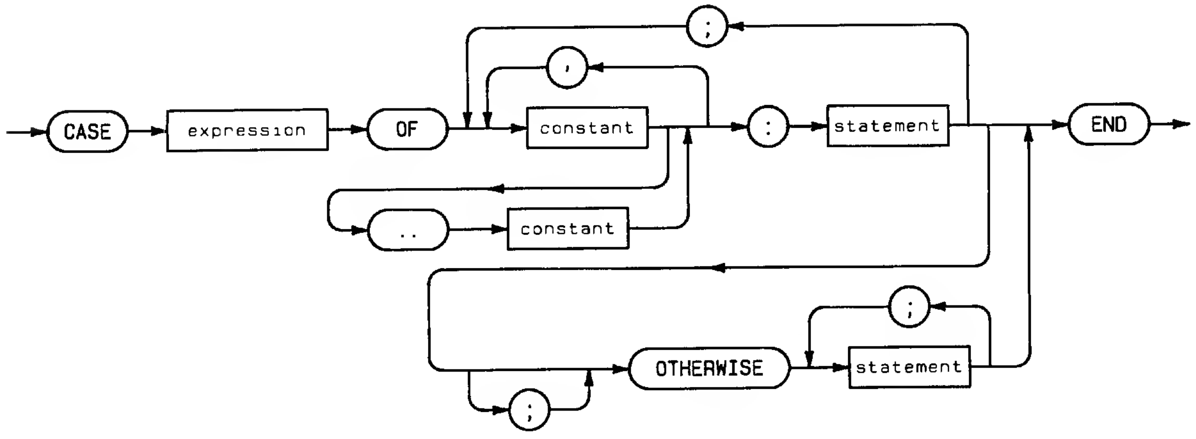
assignment:	`:=`
boolean:	`AND, OR, NOT`
relational:	`<, <=, =, <>, >=, >, IN`

Standard Functions

boolean argument:	`ord, pred, succ`
boolean return:	`eof, eoln, odd`

CASE

The CASE statement selects a certain action based upon the value of an ordinal expression.



Semantics

The CASE statement consists of the reserved word CASE, an ordinal expression (the selector), the reserved word OF, a list of case constants and statements, and the reserved word END. Optionally, the reserved word OTHERWISE and a list of statements may appear after the last constant and its statement.

The selector must be an ordinal expression, i.e. it must return an ordinal value. A case constant may be a literal, a constant identifier, or a constant expression which is type compatible with the selector. Subranges may also appear as case constants.

A case constant cannot appear more than once in a list of case constants. Subranges used as case constants may not overlap other constants or subranges.

Several constants may be associated with a particular statement by listing them separated by commas.

You need not bracket the statements between OTHERWISE and END with BEGIN..END.

When the system executes a CASE statement:

1. It evaluates the selector.
2. If the value corresponds to a specified case constant, it executes the statement associated with that constant. Control then passes to the statement following the CASE statement.
3. If the value does not correspond to a specified case constant, it executes the statements between OTHERWISE and END. Control then passes to the statement after the CASE statement. A run time error occurs if you have not used the OTHERWISE construction.

Example Code

```
PROCEDURE scanner;
  BEGIN
    get_next_char;
    CASE current_char OF
      'a'..'z',                   {Subrange label. }
      'A'..'Z':
          scan_word;

      '0'..'9':
         scan_number;

      OTHERWISE scan_special;
    END;
  END;
. . . .

FUNCTION octal_digit
   (d: digit): boolean;   {TYPE digit = 0..9}
  BEGIN
    CASE d OF
      0..7: octal_digit := true;
      8..9: octal_digit := false;
    END;
  END;
. . . .

FUNCTION op   {TYPE operators=(plus,minus,times,divide)}
  (operator: operators;
   operand1,
   operand2: real)
  : real;
  BEGIN
    CASE operator OF
      plus:   op := operand1 + operand2;
      minus:  op := operand1 - operand2;
      times:  op := operand1 * operand2;
      divide: op := operand1 / operand2;
    END;
   END;
```

char

This predefined ordinal type is used to represent individual characters.

The `char` type allows the 8-bit ASCII character set.

A pair of single quote marks encloses a `char` literal.

Permissible Operators

assignment:	`:=`
relational:	`<, <=, =, <>, >=, >, IN`

Standard Functions

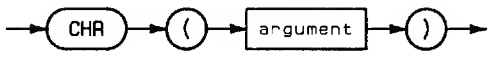
char argument:	`ord`
char return:	`chr, pred, succ`

Example Code

```
VAR
  do_you:  char;

BEGIN
  do_you := 'Y';
END;
```

chr

This function converts an integer numeric value into an ASCII character.



Item	Description/Default	Range Restrictions
argument	integer numeric expression	0 thru 255

Examples

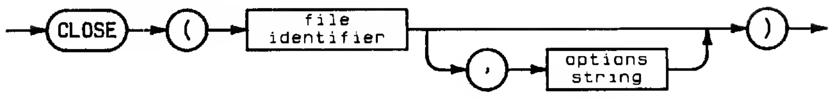
Input	Result
`chr(x)`	
`chr(63)`	`'?'`
`chr(82)`	`'R'`
`chr(13)`	(carriage return)

Semantics

The function `chr`(x) returns the character value, if any, whose ordinal number is equal to the value of x. An error occurs if x is not within the range 0..255.

close

This procedure closes a file from further access.



Item	Description/Default	Range Restrictions
file identifier	name of a logical file	-
options string	a string expression or PAC variable	implementation dependent

Examples

```
close(fil_var)
close(fil_var,opt_str)
```

Semantics

The procedure close(f) closes the file f so that it is no longer accessible. After close, references to the function eof(f) or the buffer variable (f^) will result in an error, and any association of f with a physical file is dissolved.

When closing a direct access file, the last component of the file will be the highest-indexed component ever written to the file (lastpos(f)). The value of maxpos for the file, however, remains unchanged.

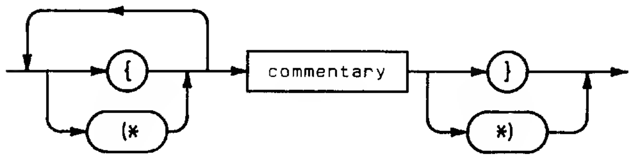
Once a file is closed, it may be reopened. Any other file operation on that file will produce an error.

Option String

The options string specifies the disposition of any physical file associated with the file. The value is implementation dependent. The compiler ignores leading and trailing blanks and considers upper and lower case equivalent. If no options string is supplied, the file retains its previous (original) status.

Comments

Comments consist of a sequence of characters delimited by the special symbols { and }, or the symbols (* and *). The compiler ignores all the characters between these symbols. Comments usually document a program.



Examples

```
{comment}
(*comment*)
{comment*)
{ { { {comment}
{This comment
 occupies more than one line.}
```

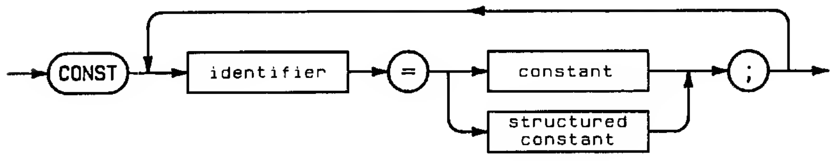
Semantics

A comment is a separator and may appear anywhere in a program a separator may appear. A comment may begin with { and close with *), or begin with (* and close with }.

Nested comments are not legal, however, a comment may cross a line boundary in source code.

CONST

This reserved word indicates the beginning of one or more constant definitions.



Semantics

Constant definitions appear after the program header (any LABEL declarations) and before any procedure or function definitions. In HP Pascal, CONST, TYPE, and VAR definitions may be intermixed.

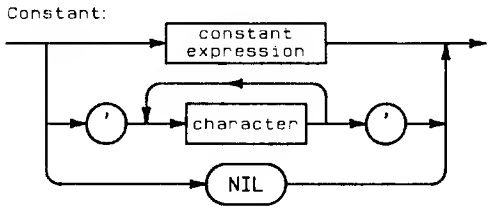
Example Code

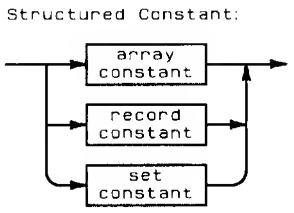
```
PROGRAM show_CONST;

LABEL 1;

 TYPE
   type1 = integer;
   type2 = boolean;
   str1  = string[5];

CONST
   const1 = 3.1415;
   const2 = true;
   strconst = str1['abcde'];

VAR
   var1 : type1;

BEGIN
END.
```

Constants

A constant definition establishes an identifier as a synonym for a constant value. The identifier may then be used in place of the value. The value of a symbolic constant may not be changed by a subsequent constant definition or by an assignment statement.

The reserved word CONST precedes one or more constant definitions. A constant definition consists of an identifier, the equals sign (=), and a constant value. (See CONST.)





The reserved word NIL is a pointer value representing a nil-value for all pointer types. Declared constants include the standard constants `maxint` and `minint` as well as the standard enumerated constants `true` and `false`.

Constant expressions are a restricted class of HP Pascal expressions. They must return an ordinal value which is computable at compile time. Consequently, operands in constant expressions must be integers or ordinal declared constants. Operators must be +, −, *, DIV, or MOD. All other operators are excluded. Furthermore, only calls to the standard functions `ord`, `chr`, `pred`, `succ`, `abs`, `hex`, `octal`, and `binary` are legal.

Floating-point values are not allowed in constant expressions.

One exception to the restrictions on constant expressions is permitted: you may change the sign of a real or longreal declared constant using the negative real unary operator (−). The positive operator (+) is legal but has no effect.

A constructor specifies values for a previously declared array, `string`, record, or set type. Subsequent pages describe constructors and the structured declared constants they define.

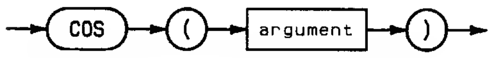
Constant definitions must follow label declarations and precede function or procedure declarations. You can repeat and intermix CONST sections with TYPE and VAR sections.

Example Code

```
CONST
  fingers   = 10;             {Unsigned integer.          }

  pi        = 3.1415;         {Unsigned real.             }

  message   = 'Use a fork!';  {String literal.            }

  nothing   = NIL;

  delicious = true;           {Standard constant.         }

  neg_pi    = -pi;            {Real unary operator.       }

  hands     = fingers DIV 5;  {Constant expression.       }

  numforks  = pred(hands);    {Constant expression with   }
                              {call to standard function. }
```

COS

This function returns the cosine of the angle represented by its argument (interpreted in radians). The range of the returned value is -1 thru $+1$.



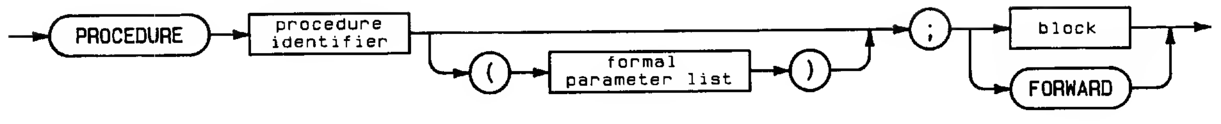
Item	Description/Default	Range Restrictions
argument	numeric expression	implementation dependent

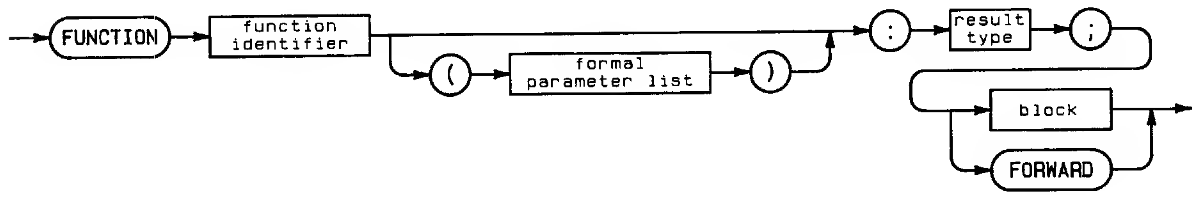
Examples

Input	Result
`cos(x_rad)`	
`cos(1.62)`	`-4.91836E+00`

Directives

A directive may replace a block in a procedure or function declaration.





In HP Standard Pascal, the only directive is FORWARD. The FORWARD directive makes it possible to postpone full declaration of a procedure or function. Additional directives may be provided by an implementation.

The term FORWARD may appear as an identifier in source code and, at the same time, as a directive.

FORWARD Directive

The FORWARD directive permits the full declaration of a procedure or function to follow the first call of the procedure or function. For example, suppose you declare procedures A and B on the same level. Both A and B cannot call each other without using the FORWARD directive.

```
PROCEDURE A; FORWARD;
PROCEDURE B;
  BEGIN
    .
    A;    {calls A}
    .
  END;
PROCEDURE A;  {full declaration of A}
  BEGIN
    .
    B;    {calls B}
    .
  END;
```

After using the FORWARD directive, you must fully declare the function or procedure in the same declaration part of the block. Formal parameters, if any, and the function result type must appear with the FORWARD declaration. You may omit these formal parameters or result type, however, when making the subsequent full declaration (see example below). If repeated, they must be identical with the original formal parameters or result type.

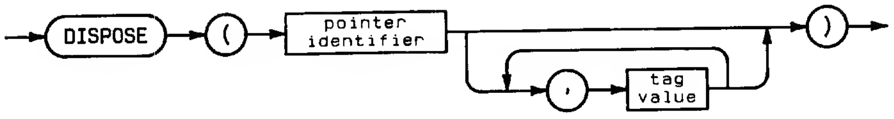
The FORWARD directive may appear with a procedure or function at any level.

Example Code

```
FUNCTION exclusive_or (x,y: boolean): boolean;
  FORWARD;
  .
  .
FUNCTION exclusive_or;         {Parameters not repeated.}
  BEGIN
    exclusive_or:= (x AND NOT y) OR (NOT x AND y);
  END;
```

dispose

This procedure indicates that the storage allocated for the given dynamic variable is no longer needed.



Item	Description/Default	Range Restrictions
pointer identifier	a variable of type pointer	cannot be NIL or undefined
tag value	a case constant value	must match case constant value specified in `new`

Examples

```
dispose(ptr_var)
dispose(ptr_var, t1,...,tn)
```

Semantics

The procedure `dispose`(p) indicates that the storage allocated for the dynamic variable referenced by p is no longer needed.

An error occurs if p is NIL or undefined. After `dispose`, the system has closed any files in the disposed storage and p is undefined.

If you specified case constant values when calling `new`, the identical constants must appear as t parameters in the call to `dispose`.

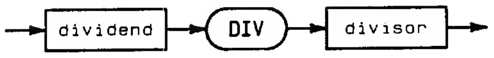
The pointer p must not reference a dynamic variable which is currently an actual variable parameter, an element of the record variable list of a WITH statement, or both.

Example Code

```
PROGRAM show_dispose (output);
TYPE
  marital_status = (single, engaged, married, widowed, divorced);
  year = 1900..2100;
  ptr  = ^person_info;
  person_info = RECORD
                  name: string[25];
                  birdate: year;
                  next_person: ptr;
                  CASE status: marital_status OF
                    married..divorced: (when: year;
                                        CASE has_kids: boolean OF
                                          true: (how_many:1..50);
                                          false: ()
                                        ););
                    engaged: (date: year)
                    single : 1;
                END;
VAR
  p : ptr;
BEGIN
  .
  .
  new(p);
  .
  .
  dispose(p);
  .
  .
  new(p,engaged);
  .
  .
  dispose(p,engaged);
  .
  .
  new(p,married,false);
  .
  .
  dispose(p,married,false);
  .
  .
END.
```

DIV

This operator returns the integer portion of the quotient of the dividend and the divisor.



Item	Description/Default	Range Restrictions
dividend	an integer or integer subrange	-
divisor	an integer or integer subrange	not equal to 0

Examples

```
Input                          Result
dvd DIV dvr
413 DIV 6                       68
```

DO

See FOR, WHILE, WITH.

DOWNTO

See FOR.

ELSE

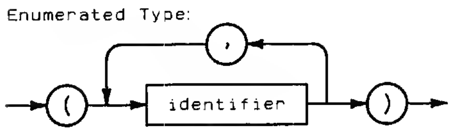
See IF.

END

See BEGIN.

Enumerated Types

An enumerated type is an ordered list of identifiers in parentheses. The sequence in which the identifiers appear determines the ordering. The `ord` function returns 0 for the first identifier; 1 for the second identifier; 2 for the third identifier; and so on.



There is no arbitrary limit on the number of identifiers that may appear in an enumerated type. The limit is implementation dependent.

Enumerated types are user-defined simple ordinal types.

Permissible Operators

assignment:	`:=`
relational:	`<, <=, =, <>, >=, >, IN`

Standard Functions

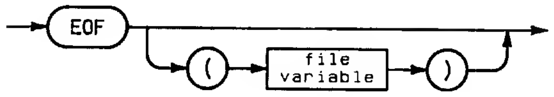
enumerated argument:	`ord, pred, succ`
enumerated return:	`pred, succ`

Example Code

```
TYPE
  days = (monday, tuesday, wednesday,
          thursday, friday, saturday, sunday);
 color = (red, green, blue, yellow, cyan, magenta, white, black);
```

eof

This boolean function returns `true` when the end of a file is reached.



Item	Description/Default	Range Restrictions
file variable	variable of type `file`	file must be open

Examples

```
eof
eof(file_var)
```

Semantics

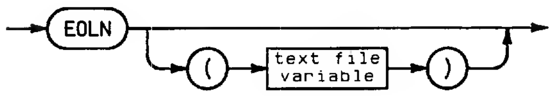
If the file f is open, the boolean function `eof`(f) returns `true` when f is in the write-only state, when f is in the direct access state and its current position is greater than the highest-indexed component ever written to f, or when no component remains for sequential input. Otherwise, `eof`(f) returns `false`. If `false`, the next component is placed in the buffer variable.

When reading non-character values (e.g. `integers`, `reals`, etc.) from a textfile, `eof` may remain `false` even if no other value of that type exits in the file. This can occur if the remaining components are blanks.

If f is omitted, the system uses the standard file `input`.

eoln

This boolean function returns `true` when the end of a line is reached in a textfile.



Item	Description/Default	Range Restrictions
textfile variable	variable must be a textfile	file must be open in the read-only state

Examples

```
eoln
eoln(text_file)
```

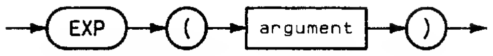
Semantics

The boolean function `eoln`(f) returns `true` if the current position of textfile f is at an end-of-line marker. The function references the buffer variable f^, possibly causing an input operation to occur. For example, after `readln`, a call to `eoln` will place the first character of the new line in the buffer variable.

If f is omitted, the system uses the standard file `input`.

exp

This real function raises e to the power of the argument. The value used for Naperian e is implementation dependent.



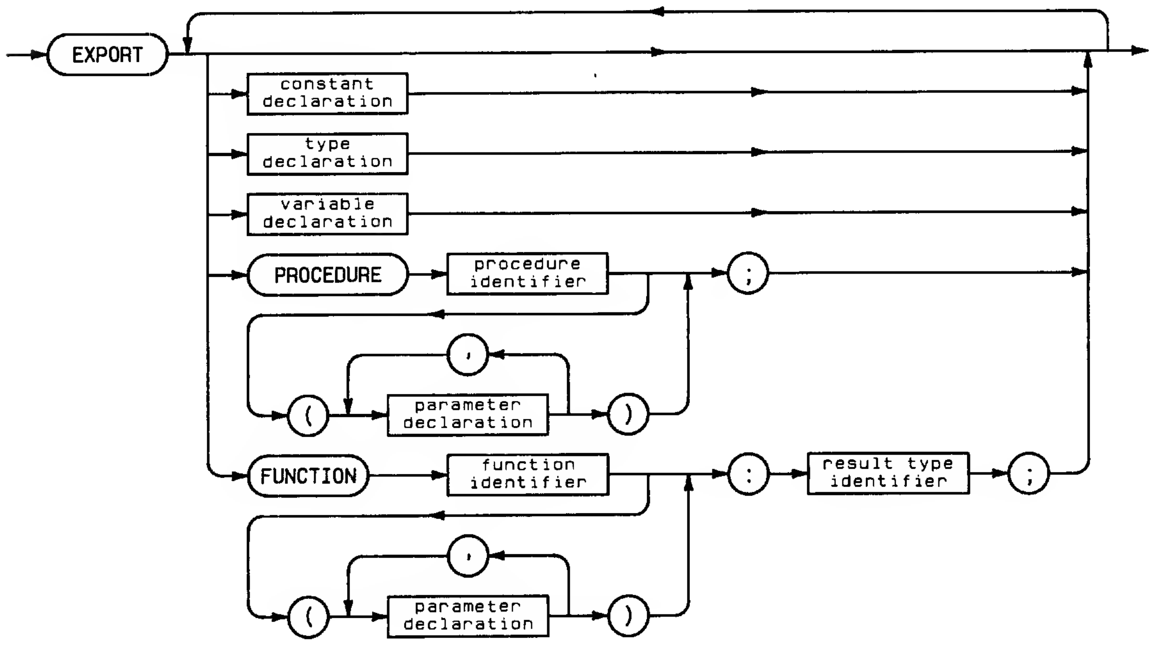
Item	Description/Default	Range Restrictions
argument	numeric expression	implementation dependent

Examples

```
Input                        Result
exp(num_exp)
exp(3)                       2.00855369231877L+001
exp(8.8E-3)                  1.008839E+00
exp(8.8L-3)                  1.00883883382898L+000
```

EXPORT

This reserved word precedes the types, constants, variables, procedures, and functions of a MODULE which can be used (IMPORTed) by other programs and modules.



See MODULE.

Expressions

An expression is a construct which represents the computation of a result of a particular type. An expression is composed of operators and operands. An operator performs an action on objects denoted by operands and produces a value.

Operators are classified as arithmetic, boolean, relational, set, or concatenation operators. An operand may be a literal, constant identifier, set constructor, or variable. Function calls are also operands in the sense that they return a result which an operator can use to compute another value.

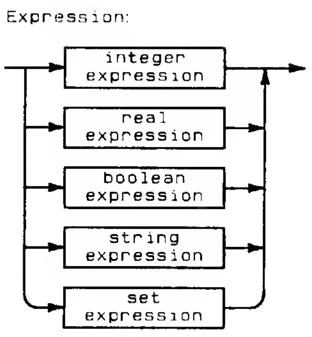
The result type of an expression is determined when the expression is written. It never changes. The actual result, however, may not be known until the system evaluates the expression at run time. It may differ for each evaluation. A constant expression is an expression whose actual result is computable at compile time.

In the simplest case, an expression consists of a single operand with no operator.

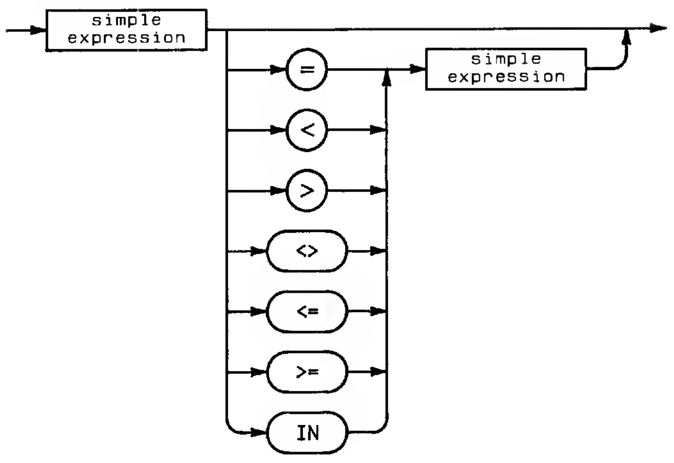
Examples

```
x:= 19;                 {Simplest case. "19" is the expression  }
                        { in the statement: "x := 19"           }

100 + x;                {Arithmetic operator with literal and   }
                        {variable operands.                     }

(A OR B) AND (C OR D)   {Boolean operator with boolean operands.}

x < y                   {Relational operator with variable      }
                        {operands.                              }

setA * setB;            {Set operator with variable operands.   }

'ice'+'cream'           {Concatenation operator with string     }
                        {literal operands.                      }
```

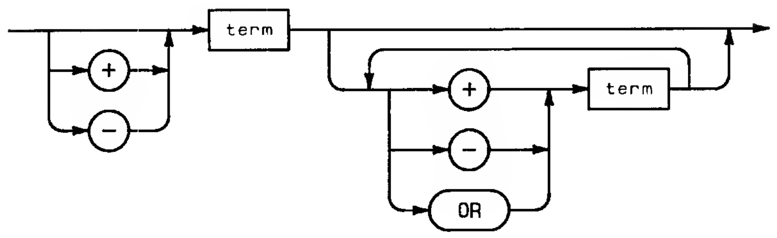
Syntax



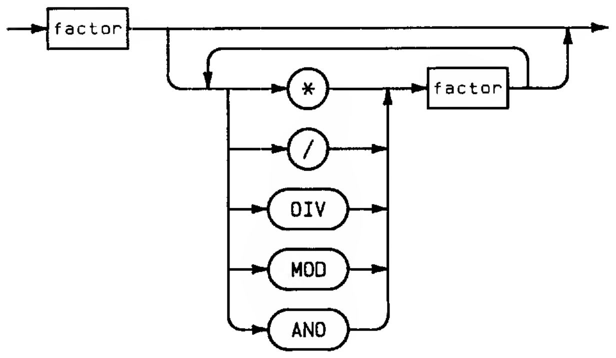
Expression



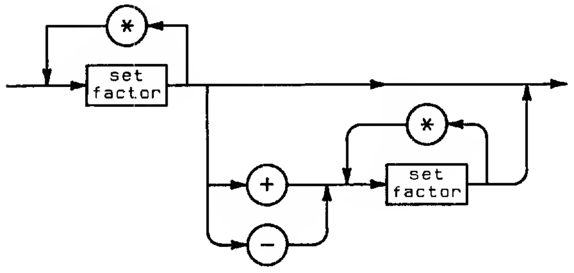
Simple Expression



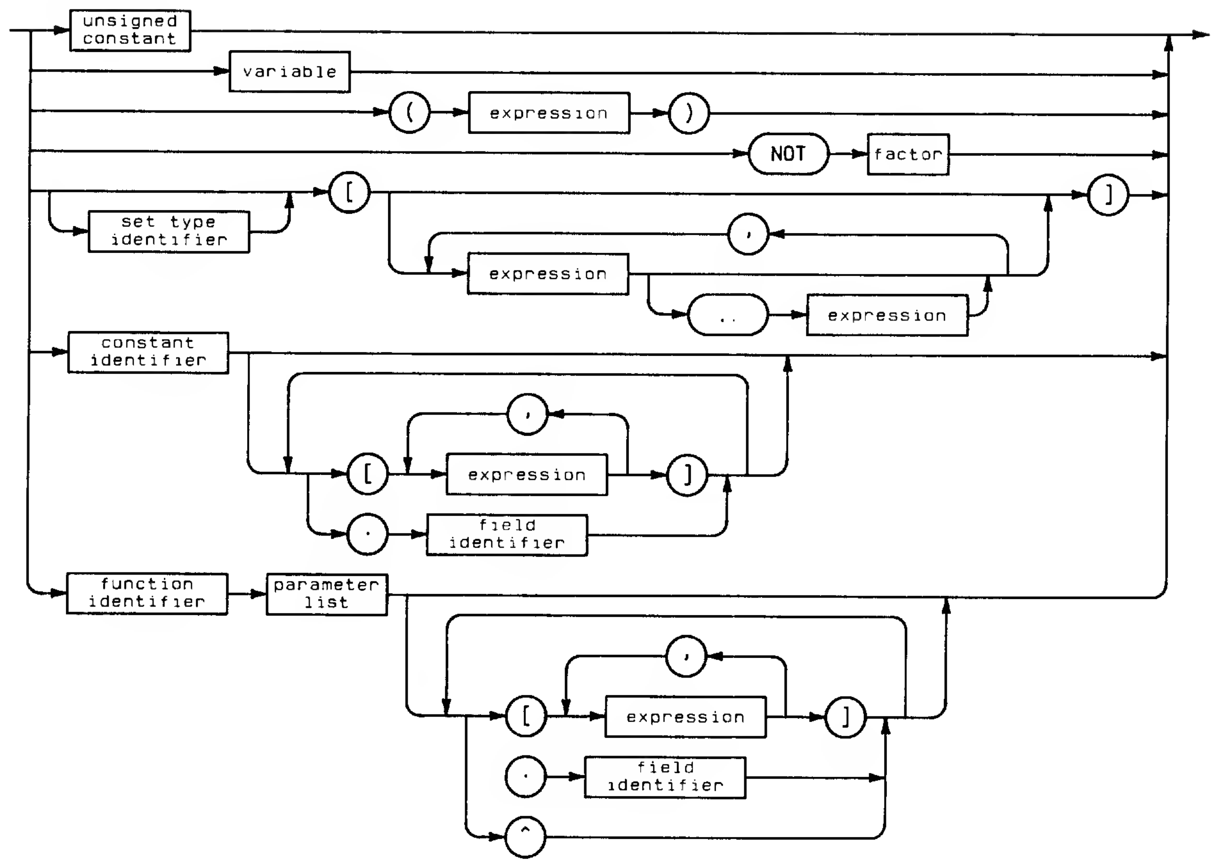
Term



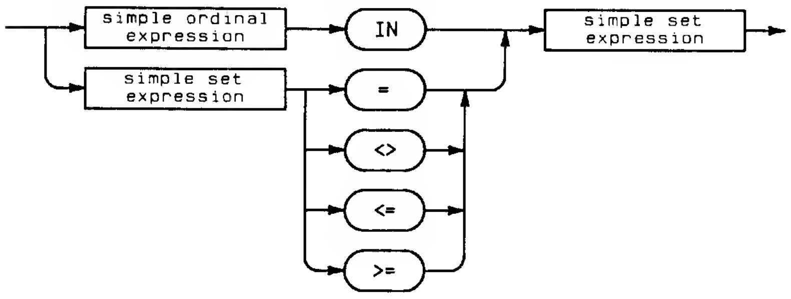
Simple Set Expression:



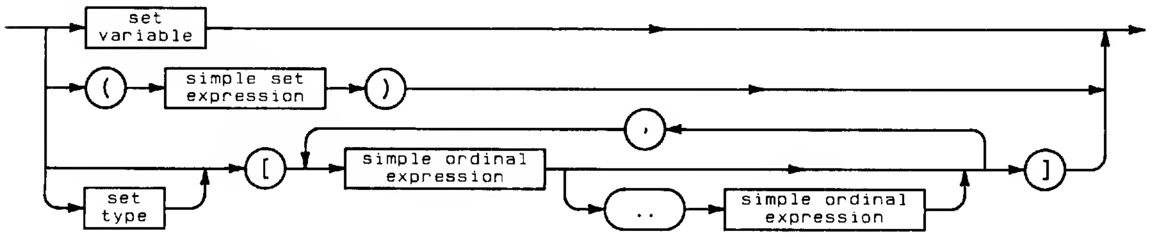
Factor



Relational Expressions Involving Sets:



Set Factor:



false

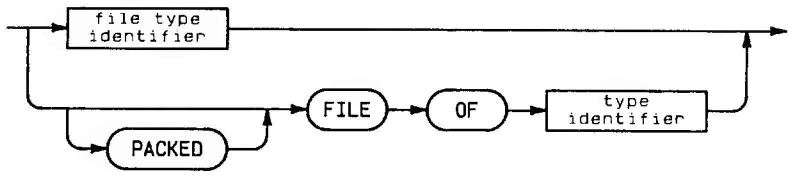
This predefined boolean constant is equal to the boolean value false.

Example Code

```
PROGRAM show_false(output);

TYPE
  what, lie : boolean;

BEGIN
  IF false THEN writeln('always false, never printed');
  what := false;
  lie := NOT true;
  IF what = lie THEN writeln('Would I lie?');
END.
```

FILE

This reserved word designates a declared data structure.



Semantics

A file type consists of the reserved words FILE OF and a component type. See also `text`.

A logical file is a declared data structure in a HP Pascal program. A physical file is an independent entity controlled by the operating system. During execution, logical files are associated with physical files, allowing a program to manipulate data in the external environment.

A logical file is a sequence of components of the same type, which may be any type except a file type or a structured type with a file type component. The number of components is not fixed by the file type definition.

File components may be accessed sequentially or directly using a variety of HP Pascal standard procedures and functions.

It is legal to declare a packed file. Whether this has any effect on the storage of the file is implementation dependent.

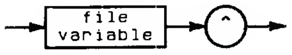
Example Code

```
TYPE
  person       = RECORD
                   name:  PACKED ARRAY [1..30] OF char;
                   age:   1..100;
                 END;
  person_file  = FILE OF person;

  bit_vector   = PACKED ARRAY [1..100] OF boolean;
  vector_file  = FILE OF bit_vector;

  data_file    = FILE OF integer;
  doc_file     = text;
```

File Buffer Selector

A file buffer selector accesses the contents, if any, of the file buffer variable associated with the current position of a file. The selector follows a file designator and consists of the caret symbol (^).

Buffer Variable:



A file designator is the name of a file or the selected component of a structure which is a file. The @ symbol may replace the caret.

If the file buffer variable is not defined at the time of selection, a run time error occurs.

Example Code

```
PROGRAM show_bufferselector;
VAR
  f   : FILE OF integer;
  a,b : integer;
BEGIN
    .
  a:= f^ + 2;              {Assigns current contents of file }
    .                      {buffer plus 2 to a.              }
    .
  f^:=a + b;               {Assigns sum of a and b to buffer }
    .                      {variable.                        }
 END.
```

Files

Files are the means by which a program receives input and produces output. A file is a sequence of components of the same type. This type may be any type, except a file type or a structured type with a file type component.

Logical files are files declared in a HP Pascal program. Physical files are files which exist independently of a program and are controlled by the operating system. You may associate logical and physical files so that a program manipulates data objects external to itself.

The components of a file are indexed starting at component 1. Each file has a current component. The standard procedure `read`(f,x) copies the contents of the current component into x and advances the current position to the next component. The procedure `write`(f,x) copies x into the current component and, like `read`, advances the current position.

Each file has a buffer variable whose contents, if defined, are accessible using a selector.

One of the standard procedures `reset`, `rewrite`, `append`, or `open` opens a file for input or output. The manner of opening a file determines the permissible operations. In particular, `reset` opens a file in the read-only state, i.e. writing is prohibited; `rewrite` and `append` open a file in the write-only state, i.e. reading is prohibited; and `open` opens a file in the read-write state, i.e. both reading and writing are legal.

All files are automatically closed on exit from the block in which they are declared, whether by normal exit or non-local GOTO. Files allocated on the heap are automatically closed when the file or structure containing the file is disposed. All files are closed at the end of the program.

Files opened with `reset`, `rewrite`, or `append` are sequential files. The current position advances only one component at a time. Files opened with `open` are direct access files. You may relocate the current position anywhere in the file using the procedure `seek`. Direct access files have a maximum number of components determinable with the standard function `maxpos`. The maximum number of components of a sequential file, on the other hand, is not determinable with a Pascal function.

Textfiles are sequential files with `char` type components. Furthermore, end-of-line markers substructure textfiles into lines. The standard procedure `writeln` creates these markers. The standard files `input` and `output` are textfiles. You cannot open textfiles for direct access.

The following table lists each HP Pascal file procedure or function together with a brief description of its action. The third column of the table indicates the permissible categories of files which a procedure or function may reference.

File Procedures and Functions

Procedure or Function	Action	Permissible Files
append	Opens file in write-only state. Current position is after last component and eof is true.	any
close	Closes a file.	any
eof	Returns true if file is write-only, if no component exists for sequential input, or if current position in direct access file is greater than lastpos.	any
eoln	Returns true if the current position of a text file is at a line marker.	read-only textfiles
get	Allows assignment of current component to buffer and, in some cases, advances current position.	read-only or read-write files
linepos	Returns number of characters read from or written to textfile since last line marker.	textfiles
lastpos	Returns index of highest written component of direct access file.	direct access files
maxpos	Returns maxint or the maximum component read or written. Check implementation.	direct access files
open	Opens file in read-write state. Current position is 1 and eof is false.	any except a textfile
overprint	A form of write which causes the next line of a textfile to print over the textfiles current line.	write-only textfiles
page	Causes skip to top of new page when a textfile is printed.	write-only textfiles
position	Returns integer indicating the current component of a non-text file.	any file except a textfile

Procedure or Function	Action	Permissible Files
prompt	A form of write which assures textfile buffers have been written to the device. No line marker is written.	write-only textfiles
put	Assigns the value of the buffer variable to the current component and advances the current position.	write-only or read-write files
read	Copies current component into specified variable parameter and advances current position.	read-only or read-write files
readdir	Moves current position of a direct access file to designated component and then performs read.	direct access files
readln	Performs read on textfile and then skips to next line.	read-only textfiles
reset	Opens file in read-only state. Current position is 1.	any
rewrite	Opens file in write-only s035tate. Current position is 1 and eof is true. Old components discarded.	any
seek	Places current position of direct access file at specified component number.	direct access files
write	Assigns parameter value to current file component and advances current position.	write-only or read-write files
writedir	Advances current position in direct access file to designed component and performs a write.	direct access files
writeln	Assigns parameter value to current textfile component, appends a line marker and advances current position.	write-only textfiles

Opening and Closing Files

A program must open a logical file before any input, output, or other file operation is legal. Four file opening procedures are available: `reset`, `rewrite`, `append`, or `open`. When they appear as program parameters, the standard textfiles `input` and `output` are exceptions to this rule. The system automatically resets `input` and rewrites `output`.

The procedure `reset` opens a file in the read-only state without disturbing its contents. After `reset`, the current position is the first component and the program can read data sequentially from the file. No output operation is possible.

The procedure `rewrite` opens a file in the write-only state and discards any previous contents. After `rewrite`, the current position is the beginning of the file. The program can then write data sequentially to the file. No input operation is possible.

The procedure `append` is identical to the procedure `rewrite` except that the current position is placed after the last component and the file contents are undisturbed. The program can then append data to the file.

The procedure `open` opens a file in the read-write state. The contents of the file, if any, are undisturbed and the current position is the beginning of the file. The program may then read or write data.

A file opened in the read-write state is a direct access file. Using the procedure `seek`, the current position can be placed anywhere in the file. Furthermore, direct access files permit calls to the standard procedures `readdir` or `writedir`, which are combinations of `seek` and the procedures `read` or `write`. Direct access files have a maximum number of components. The function `maxpos` returns this number.

In contrast, files opened in the read-only or write-only states are sequential files; the current position only advances one component at a time and the maximum number of components cannot be determined by a Pascal function.

The procedure `close` explicitly closes any logical file and its associated physical file. You need not use this procedure, however, before opening a file in a new state. For example, suppose file f is in the write-only state and the program calls `reset(f)`. The system first closes f and then resets f in the read-only state.

The system also closes a file, not on the heap, when the program exits from the scope in which the file was declared. The system closes a "heap" file when the `dispose` procedure uses the pointer to the file as a parameter or when the program terminates.

When a program finishes using an existing file, the file is closed in the same state that it existed when it was opened.

When a program closes a file it has created, the implementation may allow an optional parameter to be specified in the `close` procedure. This parameter may affect the state of the file after the program terminates.

I/O Considerations

The procedures read and write perform the fundamental input and output operations. Read(f,x) copies the contents of the current component into x and advances the current position. Write(f,x) copies x into the current component and advances the current position.

The original Pascal standard describes read and write in terms of the buffer variable f^ and the procedures get and put. The procedure put writes the contents of the buffer variable to the current component and then advances the position. The procedure get copies the current component to the buffer variable and advances the position.

Thus, the following are equivalent:

```
Write(f,x)                     f^:= x;
                               put(f);
```

And these are equivalent:

```
Read(f,x)                      x:= f^;
                                   get(f);
```

These definitions of get and read, however, have certain unfortunate consequences when I/O operations occur with interactive devices such as terminals (which were not available at the time Pascal was designed). In particular, at the initiation of a program or following a call to readln, the system tries to read a response before asking the question (writing a prompt).

HP Standard Pascal addresses this issue by defining a "deferred" get which postpones the actual loading of a component into the buffer variable. When programming, keep these practical implications in mind:

1. Suppose read(f,x) has just placed the value of component n in x. Then a reference to f^ copies the value of component n+1 into the buffer variable. It isn't necessary to call get explicitly. If get is called after a read, however, a reference to f^ copies the value of component n+2 into the buffer. Component n+1 is skipped.
2. The buffer variable is undefined after calls to put, write, seek, writedir, writeln, open, rewrite, and append. Before inspecting the current component, you must call get or read explicitly.
3. It is best not to use the buffer variable with direct access files. After read, for example, a reference to f^ places the next component in the buffer even if f^ appears on the left side of an assignment statement.
4. When reading a file sequentially, there may come a time when no component is available for assignment to x. Calling read in this case will cause a run-time error. You should use eof to determine if another component exists. On some files, notably terminals, this may require that a device read be performed to request another component. The component is held in the files's buffer variable and will be produced as the next result of a call to read.

5. If f is a direct access file, `eof(f)` is distinct from `maxpos(f)`. In particular `eof` is determined by the highest-indexed component ever written to f. `Maxpos`, on the other hand, is a limit on the size of the associated physical file. An error occurs if a program attempts to read a a component beyond the current `eof`. It is always possible, however, to write to a component with an index no greater than `maxpos(f)`. This will create a new `eof` condition if the index of the component written is greater than the index of any previously written component. It is never possible to write beyond `maxpos(f)`. See the implementation section.
6. When writing to a direct access file, a program may skip certain components. If the file is later read sequentially, these components will have unpredictable values.
7. In a direct access file, the system doesn't allocate components preceding n until n is written. If n is very large and preceded by many unused components, this allocation may take a significant amount of time. (Use lower-indexed components in preference to higher-indexed components.)

Logical Files

Any file declared in the declaration part of a HP Pascal block is a logical file. Within a program, the scope of a file name is the scope of any other HP Pascal identifier. However, you may associate the logical file with a physical file that exists outside the program. Then operations performed on the logical file are performed on the physical file.

A logical file consists of a sequence of components of the same type. This type may be any type, except the type file or a structured type with a file type component. Every logical file has a buffer variable and a current position pointer.

The buffer variable is the same type as the type of the file's components. It is denoted:

```
f^
```

where f is the designator of the logical file. You can use the buffer variable to preview the value of the current component.

The current position pointer is an integer index, starting from 1. It indicates the component that the next input or output operation will reference. The function `position` returns the value of this index, except in the case of textfiles.

After certain file operations, such as `write` with direct access files, the buffer variable is undefined. You must call `get` before f^ will access the value of the current position. After other operations, such as `read`, a subsequent reference to f^ will successfully access the current component. No `get` is necessary.

You may assign the contents of f^ to a declared variable of the appropriate type. Alternatively, the value of an expression with an appropriate result type may be assigned to f^.

Textfiles are a special class of logical files substructured into lines (see below). `Input` and `output` are standard textfiles.

You must explicitly open any logical file before performing a file operation, except for `input` and `output` when they appear as program parameters (see below). The four file opening procedures are `reset`, `rewrite`, `append`, and `open` (see below). The manner of opening a logical file determines its "state". For example, a file opened with `append` is in the write-only state. No input operation is possible.

You may use the procedures `read`, `write`, `get`, and `put`, and the function `eof`, with any appropriately opened logical file, regardless of its type.

Example Code

```
PROGRAM show_logfile (input,output,bfile);
TYPE
  book_info = RECORD
    title  : PACKED ARRAY [1..50] OF char;
    author : PACKED ARRAY [1..50] OF char;
    number : 1..32000;
    status : (on_shelf,checked_out,lost,ordered)
  END;
 VAR
  old_book: book_info;
  bfile   : FILE OF book_info;   {Declaring a logical file. }
  posnum  : integer;
 BEGIN
  .
  .
  reset(bfile);   {Opening logical file which is associated }
  .               {by default with the file named 'BFILE'.  }
  .
  .
  old_book:= bfile^;           {Assigning buffer variable to }
  .                            {declared variable.           }
  .
  posnum:= position(bfile);        {Using index of current }
  .                                {component.             }
  .
END.
```

Physical Files

The operating system controls physical files which exist independently of an HP Pascal program. These files may be permanent files on disc or other media, or interactive files created at a terminal.

A particular physical file may be associated with a logical file declared in an HP Pascal program. The type of the logical file determines the characteristics of the physical file.

Except for textfiles, all physical files associated with Pascal logical files are fixed length binary files. The record length of these files depends on the type of the component.

The system associates textfiles with variable length ASCII files.

Textfiles

Textfiles are a special class of logical files which are substructured into lines by end-of-line markers. Textfiles are declared with the standard identifier `text`. The components of a textfile are type `char`.

If the current position in a textfile advances to a line marker (i.e. beyond the last character of a line), the function `eoln` returns `true` and the buffer variable is assigned a blank. When the current position advances once more, a reference to the buffer variable will access the first character of the next line and `eoln` returns `false`, unless the next line has no characters. An end-of-line marker is not an element of type `char`. Only the procedure `writeln` places it in a textfile. A line marker always precedes an eof condition, whether the last line was terminated with `writeln` or not.

The procedures `readln`, `writeln`, `page`, `prompt`, and `overprint`, and the functions `eoln` and `linepos` are available exclusively for textfiles.

Reading from a textfile may entail implicit data conversion. In certain cases, the operation searches the textfile for a sequence of characters which satisfies the syntax for a `string`, PAC, or simple type other than `char`.

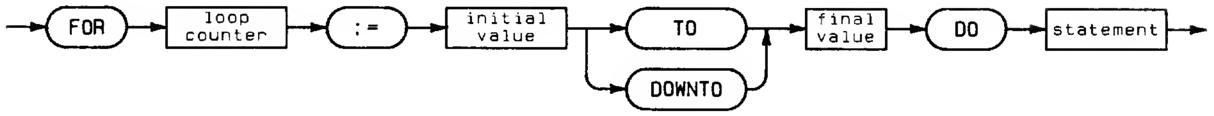
Writing to a textfile may entail formatting of the output value. You can specify a field-width parameter or allow the system to use various default field-width values.

Textfiles cannot be opened for direct access. Their format is incompatible with certain direct access operations.

The system defines two standard textfiles, `input` and `output`.

FOR

The FOR statement executes a statement a predetermined number of times.



Item	Description/Default	Range Restrictions
loop counter	ordinal variable	must be local to the block in which the loop appears
initial value	ordinal expression	-
final value	ordinal expression	-

Semantics

The FOR statement consists of the reserved word FOR and a control variable initialized by an ordinal expression (the initial value); either the reserved word TO indicating an increment or the reserved word DOWNTO indicating a decrement; another ordinal expression (the final value); the reserved word DO; and a statement.

The control variable is assigned each value of the range before the corresponding iteration of the statement.

The control variable must be a local ordinal variable. It may not be a component of a structured variable or a locally declared procedure or function parameter. The initial and final values must be type compatible with the control variable. They must also be in range with the control variable when the initial value is first assigned. The statement after DO, of course, may be a compound statement.

When the system executes a FOR statement, it evaluates the initial and final values and assigns the initial value to the control variable. Then it executes the statement after DO. Next, it repeatedly tests the current value of the control variable and the final value for inequality, increments or decrements the control variable, and executes the statement after DO.

After completion of the FOR statement, the control variable is **undefined**.

In a FOR..TO construction, the system never executes the statement after DO if the initial value is greater than the final value. In a FOR..DOWNTO construction, it never executes the statement if the initial value is less than the final value.

The FOR statement

```
FOR control_var := initial TO final DO
   statement
```

is equivalent to the statement

```
BEGIN
  temp1 := initial;
  temp2 := final;
  IF temp1 <= temp2 THEN
   BEGIN
     control_var := temp1;
     statement;
     WHILE control_var <> temp2 DO
       BEGIN
         control_var := succ(control_var);  {increment}
         statement;
       END;
   END
  ELSE BEGIN END;       {Don't execute statement at all;}
END                     {control_var now undefined.     }
```

The FOR statement

```
FOR control_var := initial DOWNTO final DO
   statement
```

is equivalent to the statement

```
BEGIN
  temp1 := initial;
  temp2 := final;
  IF temp1 >= temp2 THEN
    BEGIN
      control_var := temp1;
      statement;
      WHILE control_var <> temp2 DO
        BEGIN
          control_var := pred(control_var); {decrement}
          statement;
        END;
    END
  ELSE BEGIN END;          {Don't execute statement at all;}
END                        {control_var now undefined.     }
```

In the statement after DO, the compiler protects the control variable from assignment. You cannot pass the control variable as a variable parameter or use it as the control variable of a second FOR statement nested within the first. Furthermore, it may not appear as a parameter for the standard procedures `read` or `readln`. Also, the statement cannot call a procedure or function which changes the value of the control variable.

The system determines the range of values for the control variable by evaluating the two ordinal expressions once, and only once, before making any assignment to the control variable. So the statement sequence

```
i := 5;
 FOR i := pred(i) TO succ(i) DO writeln('i=',i:1);
```

will write

```
i=4
 i=5
 i=6
```

instead of

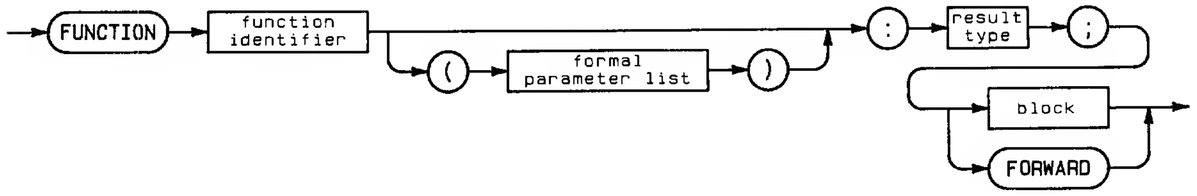
```
i=4
 i=5
```

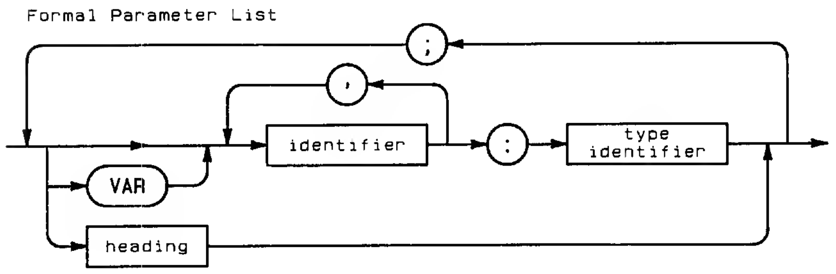
Example Code

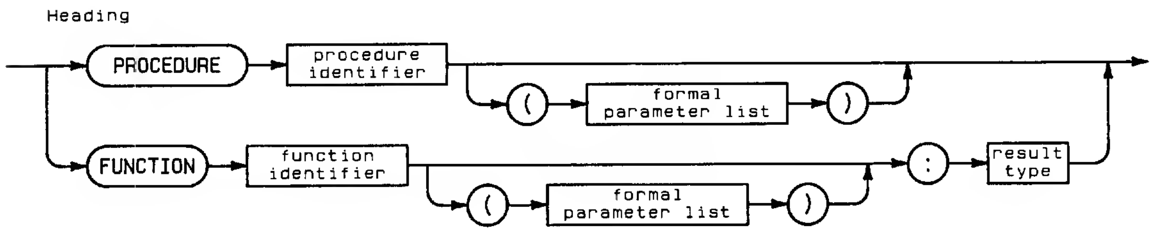
```
{VAR color: (red, green, blue, yellow);}
FOR color := red TO blue DO
  writeln ('Color is ', color);
   .
    .
 FOR i := 10 DOWNTO 0 DO
  writeln (i);
writeln ('Blast Off');
   .
   .
 FOR i := (a[j] * 15) TO (f(x) DIV 40) DO
  IF odd(i) THEN
    x[i] := cos(i)
   ELSE
    x[i] := sin(i);
```

FUNCTION

A function is a block which is activated with a function call and which returns a value. A function declaration consists of a function heading followed by a block or a directive.







Item	Description/Default	Range Restrictions
function identifier	name of a user-defined function	any valid identifier
formal parameter list	see diagram	-
result type	type identifier	any previously defined type
heading	see drawing	-

Semantics

A function heading consists of the reserved word FUNCTION, an identifier (function name), an optional formal parameter list, and a result type. The result type may be any type, except a file type or a structured type containing a file.

A directive can replace the function block to inform the compiler of the location of the block.

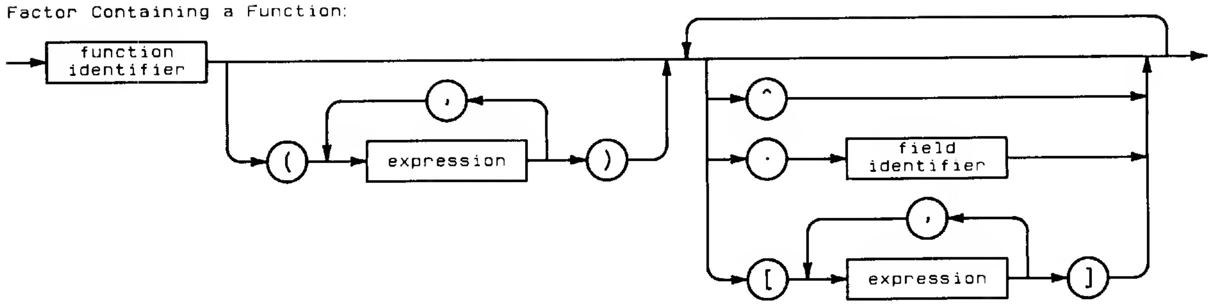
In the body of a function block there must be at least one statement assigning a value to the function identifier. This assignment statement determines the function result. If the function result is a structured type, you must assign a value to each of its components using an appropriate selector.

Function declarations may occur at the end of a declaration section after label, constant, type, variable declarations, and MODULE declarations at the outer level. You may repeat function declarations and intermix them with procedure declarations.

Function Calls

A function call activates the block of a standard or declared function.

Factor Containing a Function:



Semantics

The function returns a value to the calling point of the program. An operator can perform some action on this value and, for this reason, a function call is an operand.

A function call consists of a function identifier, an optional list of actual parameters in parentheses, and an optional selector.

The actual parameters must match the formal parameters in number, type, and order. The function result has the type specified in the function heading.

Actual value parameters are expressions which must be assignment compatible with the formal value parameters.

Actual variable parameters are variables which must be type identical with the formal variable parameters. Components of a packed structure may not appear as actual variable parameters.

Actual procedure or function parameters are the names of declared procedures or functions. Standard functions or procedures are not legal actual parameters.

The parameter list, if any, of an actual procedure or function parameter must be congruent with the parameter list of the formal procedure or function parameter. See the Procedure Statement.

Functions may call themselves recursively. See Recursion.

If an actual function or procedure parameter, upon activation, accesses any entity non-locally, then the entity accessed is one which was accessible to the function or procedure when its identifier was passed. For example, suppose Procedure A uses the non-local variable x. If A is passed as a parameter to Function B, then it still has access to x, even if x is otherwise inaccessible in B.

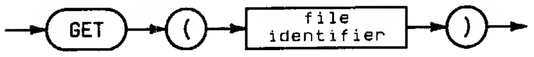
If the function result is a structured type, then the function call may select a particular component as the result. This requires the use of an appropriate selector.

Example Code

```
PROGRAM show_function (input,output);
VAR
   n,
   coef,
  answer: integer;

FUNCTION fact (p: integer) : integer;
  BEGIN
    IF p > 1 THEN
      fact := p * fact (p-1)
    ELSE fact := 1
   END;

FUNCTION binomial_coef (n, r: integer) : integer;
  BEGIN
    binomial_coef := fact (n) DIV (fact (r) * fact (n-r))
  END;


BEGIN {show_function}
  read(n);
  FOR coef := 0 TO n DO
    writeln (binomial_coef (n, coef));
END.  {show_function}
```

get

This procedure assigns the value of the current component of a file to its argument.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	file must be open to read

Example

```
get(file_var)
```

Semantics

The file must be in the read-only or read-write state.

The procedure `get`(f) advances the current file position and causes a subsequent reference to the buffer variable f^ to actually load the buffer with the current component. In certain circumstances, namely after a call to `read`, `get` also advances the current position.

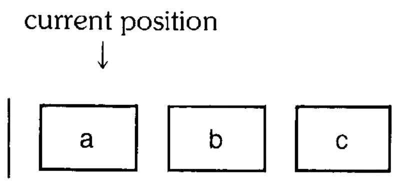
If the current component does not exist when `get` is called, f^ will be undefined and `eof`(f) will return `true`. An error occurs if f is in the write-only state or if `eof`(f) is `true` prior to the call to `get`.

If you `open` a file, a `get` must be performed before the buffer variable contains valid data. However, if you `reset` a file, the buffer variable contains valid data and a `get` should not be performed until you want to access the second component.

Illustration

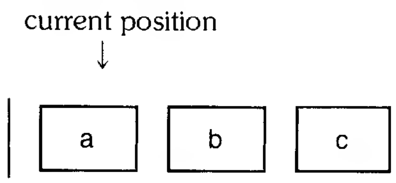
Suppose `examp_file` is a file of `char` with three components which has just been opened in the read-write state. The current position is the first component and examp_file^ is undefined. To inspect the first component, we call `get`:

{initial condition for open}



state : read-write
examp_file^: undefined
eof(examp_file) : false

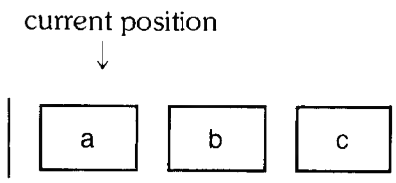
get(examp_file);



state : read-write
a b c examp_file^(deferred) : a
eof(examp_file) : false

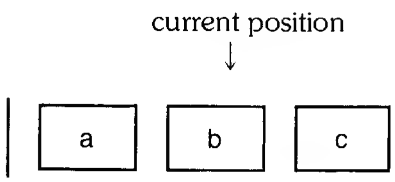
The current position is unchanged. Now, however, a reference to `examp_file^` loads the first component into the buffer. We assign the buffer to a variable.

char_var: = examp_file^



state : read-write
examp_file^ : a
eof(examp_file) : false

get(examp_file);



state : read-write
examp_file^(deferred) : b
eof(examp_file) : false

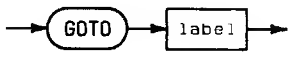
Global Variables

Global variables are declared in the outermost block of a program or module and are available to all of the procedures and functions within the program or module.

Conversely, "local" variables are declared within a particular procedure or function and their "scope" is limited to that procedure or function.

GOTO

A GOTO statement transfers control unconditionally to a statement marked by a label.



Semantics

A GOTO statement consists of the reserved word GOTO and the specified label.

The scope of labels is restricted. Labels may only mark statements appearing in the executable portion of the block where they are declared. They cannot mark statements in inner blocks. GOTO statements, however, may appear in inner blocks and reference labels in an outer block. Thus, it is possible to jump out of a procedure or function but not into one.

A GOTO statement may not lead into a component statement of a structured statement from outside that statement or from another component statement of that statement. For example, it is illegal to branch to the ELSE part of an IF statement from either the THEN part, or from outside the IF statement.

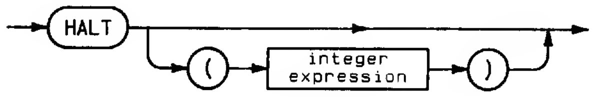
A GOTO statement which refers to a non-local label declared in an outer routine will cause any local files to be closed.

Example Code

```
PROGRAM show_goto;
LABEL 500, 501;
 TYPE
   index = 1..10;
 VAR
   i: index;
   target: integer;
   a: ARRAY[index] OF integer;
PROCEDURE check;
  VAR
    answer: string [10];
  BEGIN
    .
    {ask user if OK to search}
    IF answer= 'no' THEN GOTO 501; {Jumping out of procedure}
    .
  END;

BEGIN {show_goto}
   .
  check;
   .
  FOR i := 1 TO 10 DO
    IF target = a[i] THEN GOTO 500;
  writeln (' Not found');
  GOTO 501;
500:
  writeln (' Found');
501:
END.  {show_goto}
```

halt

This procedure terminates the execution of the program.



Examples

```
halt
halt(int_exp)
```

Semantics

Execution of a program is stopped by the `halt` procedure. When an integer expression is included, the operating system will return the integer value in an error message.

Heap Procedures

HP Pascal distinguishes two classes of variables: static and dynamic.

A static variable is explicitly declared in the declaration part of a block and may then be referred to by name in the body. The compiler allocates storage for this variable on the stack. The system does not deallocate this space until the process closes the scope of the variable.

On the other hand, a dynamic variable is not declared and cannot refer to by name. Instead, a declared pointer references this variable. The system allocates and deallocates storage for a dynamic variable during program execution as a result of calls to the standard procedures `new` and `dispose`. The area of memory reserved for dynamic variables is termed the "heap".

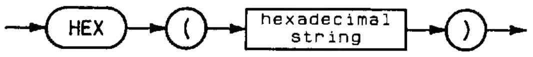
HP Pascal also supports the standard procedures `mark` and `release`. `Mark` records the state of the heap. A subsequent call to `release` returns the heap to the state recorded by `mark`. Effectively, this disposes any variables allocated since the call to `mark`.

Dynamic variables permit the creation of temporary buffer areas in memory. Furthermore, since a pointer may be a component of a structured dynamic variable, it is possible to write programs with dynamic data structures such as linked lists or trees.

Depending on implementation, `mark` and `release` may not perform any action.

hex

This function converts a hexadecimal string expression or PAC into an integer.



Item	Description/Default	Range Restrictions
hexadecimal string	string expression or PAC variable	implementation dependent

Examples

```
 Input                    Result
 hex(strng)
 hex('FF')                  255
-hex('FF')                 -255
```

If your particular implementation used 32-bit 2's complement notation, the following example would also work.

```
 hex('FFFFFF01')           -255
```

Semantics

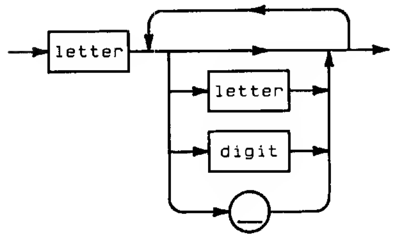
The function `hex`(s) converts s to an integer. S is interpreted as a hexadecimal value.

The three numeric conversion functions are `binary`, `hex`, and `octal`. All three accept arguments which are string or PAC variables, or string literals. The compiler ignores leading and trailing blanks in the argument. All other characters must be legal digits in the indicated base.

Since `binary`, `hex`, and `octal` return an integer value, all bits must be specified if a negative result is desired. Alternatively, you may negate the positive representation.

Identifiers

An HP Pascal identifier consists of a letter preceding an optional character sequence of letters, digits, or the underscore character (_).



Examples

```
GOOD_TIME_9         {These identifiers    }
good_time_9         {are                  }
gOOd_TIme_9         {equivalent.          }

x2_GO
a_long_identifier
boolean             {Standard identifier.}
```

Semantics

Identifiers denote declared constants, types, variables, procedures, functions, and programs.

A letter may be any of the letters in the subranges A..Z or a..z. The compiler makes no distinction between upper and lower case in identifiers. A digit may be any of the digits 0 through 9. The underscore (_) is an HP Standard Pascal extension of ANSI Standard Pascal.

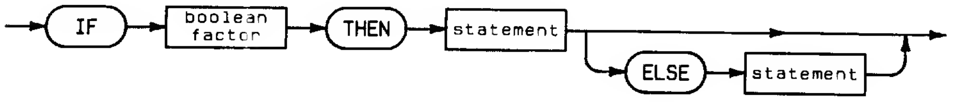
An identifier may be up to a source line in length with all characters significant.

In general, you must define an identifier before using it. Two exceptions are identifiers which define pointer types and are themselves defined later in the same declaration part, and identifiers which appear as program parameters and are declared subsequently as variables. Also, you need not define an identifier which is a program, procedure, or function name, or one of the identifiers defining an enumerated type. Its initial appearance in a function, procedure, or program header is the "defining occurrence". Finally, HP Pascal has a number of standard identifiers which may be redeclared. These standard identifiers include names of standard procedures and functions, standard file variables, standard types, and procedure or function directives.

Reserved words are system defined symbols whose meaning may never change. That is, you cannot declare an identifier which has the same spelling as a reserved word.

IF

An IF statement specifies a statement the system will execute provided that a particular condition is true. If the condition is false, then the system doesn't execute the statement, or, optionally, it executes another statement.



The IF statement consists of the reserved word IF, a boolean factor, the reserved word THEN, a statement, and, optionally, the reserved word ELSE and another statement.

When an IF statement is executed, the boolean factor is evaluated to either true or false, and one of the three actions is performed.

1. If the value is true, the statement following THEN is executed
2. If the value is false and ELSE is specified, the statement following the ELSE is executed.
3. If the value is false and no ELSE is specified, execution continues with the statement following the IF statement.

The statements after THEN or ELSE may be any HP Pascal statements, including other IF statements or compound statements. No semicolon separates the first statement and the reserved word ELSE.

The following IF statements are equivalent:

```
IF a = b THEN
  IF c = d THEN
    a := c
  ELSE
    a := e;
```

```
IF a = b THEN
  BEGIN
    IF c = d THEN
      a := c
    ELSE
      a := e;
  END;
```

That is, ELSE parts that appear to belong to more than one IF statement are always associated with the nearest IF statement.

A common use of the IF statement is to select an action from several choices. This often appears in the following form:

```
IF e1 THEN
  ...
ELSE IF e2 THEN
  ...
ELSE IF e3 THEN
  ...
ELSE
  ...
```

This form is particularly useful to test for conditions involving real numbers or string literals of more than one character, since these types are not legal in CASE statements.

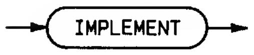
It is possible to direct the compiler to perform partial evaluation of the boolean expressions used in an IF statement. See the compiler directives for your particular implementation.

Example Code

```
PROGRAM show_if (input, output);

VAR
  i,j   :  integer;
  s     :  PACKED ARRAY [1..5] OF char;
  found:   boolean;

BEGIN
   .
   .
  IF i = 0 THEN writeln ('i = 0');   {IF with no ELSE.       }
  IF found THEN                      {IF with an ELSE part. }
    writeln ('Found it')
   ELSE
    writeln ('Still looking');
   .
   .
  IF i = j THEN                      {Select among different}
    writeln ('i = j')                {boolean expressions.   }
  ELSE IF i < j THEN
    writeln ('i < j')
  ELSE {i > j}
    writeln ('i > j');
    .
   .
  IF s = 'RED' THEN                  {This IF statement      }
    i := 1                           {cannot be rewritten as}
  ELSE IF s = 'GREEN' THEN           {a CASE statement       }
    i := 2
  ELSE IF s = 'BLUE' THEN
     i := 3;
 END.
```

IMPLEMENT

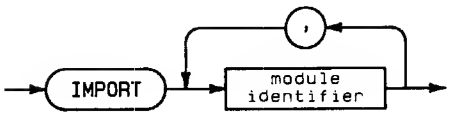
This reserved word indicates the beginning of the internal part of a MODULE. The implement section may be empty or it may contain declarations of the types, constants, variables, procedures, and functions that are only used within the module.



See MODULE.

IMPORT

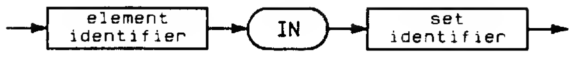
This reserved word indicates which modules will be needed to compile a program or module.



See MODULE.

IN

This operator returns true if the specified element is in the specified set.



Item	Description/Default	Range Restrictions
element identifier	expression of an ordinal type	see semantics
set identifier	expression of type SET	see semantics

Example

```
IF item IN set_of_items THEN process;
```

Semantics

Both the element being tested and the elements in the setmust be of the same type.

The result is false if the object is not a member of the set.

Example Code

```
PROGRAM show_in(output);

VAR
  ch : char;
 good : set of char;
 more : set of char;
 member : boolean;

 BEGIN
  ch := 'y';
  good := ['y','Y','n','N'];
  more := ['a'..'z'];
  IF ch IN good THEN
    member := true
  ELSE
    member := false;
  writeln(member);
END.
```

input

The standard textfiles `input` and `output` often appear as program parameters. When they do, there are several important consequences:

1. You may not declare `input` and `output` in the source code.
2. The system automatically resets `input` and rewrites `output`.
3. The system automatically associates `input` and `output` with the implementation dependent physical files.
4. If certain file operations omit the logical file name parameter, `input` or `output` is the default file. For example, the call `read`(x), where x is some variable, reads a value from `input` into x. Or consider:

```
PROGRAM mute (input);
VAR  answer : string[255];
BEGIN
  readln(answer);
END.
```

The program waits for a something to be typed. No prompt can be written without adding `output` to the program heading.

integer

This type is a subrange whose lower bound is the standard constant `minint` and whose upper bound is the standard constant `maxint`.

Examples

```
VAR
  wholenum:  integer;
  i,j,k,l :  integer;
```

Semantics

`Integer` is a standard simple ordinal type whose range is implementation defined.

Permissible Operators

assignment: `:=`

relational: `<, <=, =, <>, >, >=, IN`

arithmetic: `+, -, *, /, DIV, MOD`

Standard Functions

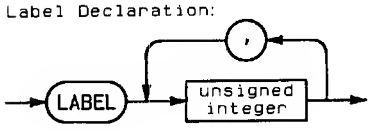
integer argument: `abs, arctan, chr, cos, exp, ln, odd, ord, pred, sin, sqr, sqrt, succ`

integer return: `abs, binary, hex, linepos, lastpos, maxpos, octal, ord, position, pred, round, strlen, strmax, strpos, sqr, trunc`

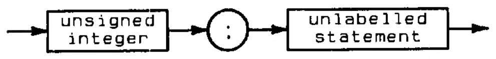
LABEL

A label declaration specifies integer labels which mark executable statements in the body of the block. The GOTO statement transfers control to a labeled statement.

Label Declaration:



Labelled Statement:



Semantics

The reserved word LABEL precedes one or more integers separated by commas.

Integers must be in the range 0 to 9999. Leading zeros are not significant. For example, the labels 9 and 00009 are identical.

Label declarations must come first in the declaration part of a block.

You cannot use a label to mark a statement in a procedure or function nested within the procedure, function, or outer block where the label is declared. This means a GOTO statement may jump out of but not into a procedure.

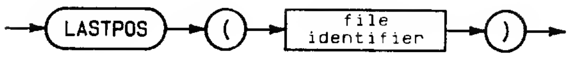
The Label declaration must occur in the declaration part of the block which contains the label.

Example

```
LABEL 9, 19, 40;
```

lastpos

This function returns the integer index of the last component of a file which has been written.



Item	Description/Default	Range Restrictions
file identifier	a file type variable	file must be opened in the read-write state

Example

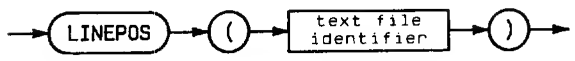
```
lastpos(file_var)
```

Semantics

The function `lastpos`(f) returns the integer index of the last component of f which the program may access. An error occurs if f is not opened as a direct access file.

linepos

This function returns the number of characters read from or written to a textfile since the last end-of-line marker.



Item	Description/Default	Range Restrictions
textfile identifier	a textfile	textfile must be opened

Example

```
linepos(text_file)
```

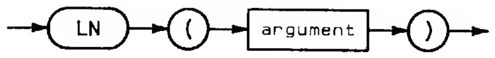
Semantics

The function `linepos`(f) returns the integer number of characters read from or written to the textfile f since the last end-of-line marker. This does not include the character in the buffer variable f^. The result is zero after reading a line marker, or immediately after a call to `readln` or `writeln`.

The standard files `input` or `output` must be specified by name.

ln

This function returns the natural logarithm (base e) of the argument.



Item	Description/Default	Range Restrictions
argument	numeric expression	must be greater than 0

Examples

```
Input                      Result
ln(num_exp)
ln(43)                     3.761200E+00
ln(2.121)                  7.518874E-01
ln(0)                      {error}
```

Semantics

The function `ln`(x) computes the natural logarithm of x. If x is 0 or less than 0, a run-time error occurs.

Local Variables

Local variables are variables declared within a particular procedure or function and their "scope" is limited to that procedure or function.

Conversely, "global" variables are declared in the outermost block of a program or module and are available to all of the procedures and functions within the program or module.

longreal

This standard simple type represents a subset of real numbers.

Semantics

`Longreal` is a standard simple type. Although similar in usage to the `real` type, the letter "L" is used to indicate the start of the exponent instead of the letter "E". (See below.)

Permissible Operators

assignment:	`:=`
relational:	`<, <=, =, <>, >=, >`
arithmetic:	`+, -, *, /`

Standard Functions

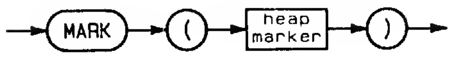
longreal argument:	`abs, arctan, cos, exp, ln, round, sin, sqr, sqrt, trunc`
longreal return:	`abs, arctan, cos, exp, ln, sin, sqr sqrt`

Example Code

```
VAR
  precisenum: longreal;
BEGIN
  precisenum:= 1.1234567891L+104;
  .
  .
```

mark

This procedure marks the state of the heap.



Item	Description/Default	Range Restrictions
heap marker	a pointer variable	-

Usage

```
mark(ptr_var)
```

Semantics

The procedure `mark`(p) marks the state of the heap and sets the value of p to specify that state. In other words, `mark` saves the state of the heap in p, which must not subsequently be altered by assignment. If altered, you will be unable to perform the corresponding release.

The pointer variable appearing as the p parameter must be a dedicated variable. It should not be dynamic variable.

`Mark` is used in conjunction with `release`. See the example under `release`.

maxint

This standard constant returns the largest value that can be represented by the integer type.

Semantics

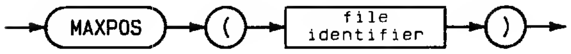
The constant maxint returns the largest value that can be represented by an integer. The value is implementation dependent.

Example Code

```
PROGRAM show_maxint(input,output);

VAR
  i,j : integer;
  r   : real;

BEGIN
  readln(i,j);
  r := i + j;
  IF r > maxint THEN writeln('Sum too large for integers.');
END.
```

maxpos

This function returns the index of the last accessible component of a file.



Item	Description/Default	Range Restrictions
file identifier	name of a logical file	file must be opened

Example

```
maxpos(file_var)
```

Semantics

The function `maxpos`(f) returns the integer index of the last component of f which the program could possibly access. An error occurs if f is not opened as a direct access (read-write) file.

For extensible files, `maxpos`(f) returns the value of `maxint`.

minint

This standard constant returns the smallest value that can be represented by the `integer` type.

Semantics

The constant `minint` returns the smallest value that can be represented by an `integer`. The value is implementation dependent.

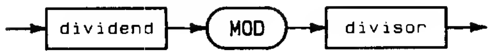
In general, the range of signed integers allows the absolute value of minint to be greater than maxint.

Example Code

```
PROGRAM show_minint(input,output);

VAR
  i,j  : integer;
  r    : real;

BEGIN
  readln(i,j); r := i - j;
  IF r < minint THEN writeln('Difference too large for integers.');
END.
```

MOD

This operator returns the remainder of an integer division.



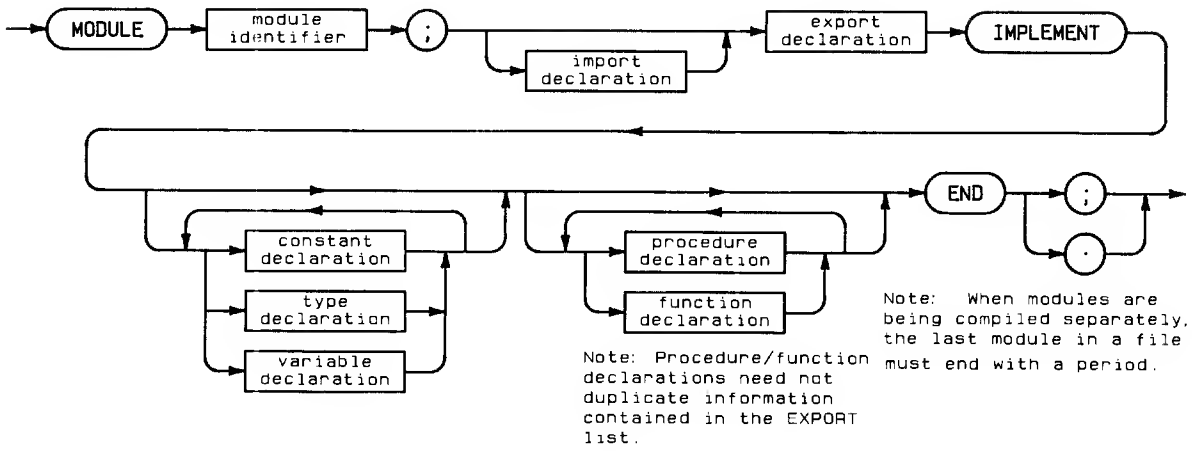
Item	Description/Default	Range Restrictions
dividend	an integer or integer subrange	-
divisor	an integer or integer subrange	greater than 0

Examples

Input	Result
`dvs MOD dvr`	
`4 MOD 3`	1
`7 MOD 5`	2

MODULE

This reserved word indicates the beginning of a separate unit of compilation.


MODULE
module identifier
;
import declaration
export declaration
IMPLEMENT
constant declaration
type declaration
variable declaration
procedure declaration
function declaration
END
;
.
Note: Procedure/function declarations need not duplicate information contained in the EXPORT list.
Note: When modules are being compiled separately, the last module in a file must end with a period.

Example

```
MODULE mod_id
```

Semantics

A MODULE can be compiled separately or included in the compilation of a program. The general form of a module is shown in the following example.

Example Code

```
MODULE show_module;                        {Module declaration          }

IMPORT my_module;                          {Other modules needed for    }
                                           {compilation of this module }

EXPORT                                     {Start of export text        }

  TYPE
    byte = 0..255;                         {Exported type               }

  VAR
    testbyte : byte;                       {Exported variable           }

  FUNCTION  control(i : byte) : boolean;  {Exported function           }

IMPLEMENT                                  {Start of implementation     }

  TYPE
    boot = 0..255;                         {Non-exported type           }

 PROCEDURE check(i : byte);               {Non-exported procedure      }
    BEGIN
      IF i > 127 THEN writeln('non-ASCII character');
    END;

  FUNCTION control(i :byte) : boolean;     {Exported function           }
    BEGIN
      IF i < 32 then control := true
      ELSE control := false;
    END;

END.
```

Modules

A module provides a mechanism for separate compilation of program segments.

Semantics

A module is a program fragment which can be compiled independently and later used to complete otherwise incomplete programs. A module usually defines some data types and variables, and some procedures which operate on these data. Such definitions are made accessible to users of the module by its export declarations.

The source text input to a compiler (complete unit of compilation) may be a program or a list of modules separated by semicolons (;). An implementation may allow only a single module to be compiled at a time, thus requiring multiple invocations of the compiler to process several modules. The input text is terminated by a period.

A module is a collection of global declarations which may be compiled independently and later made part of a program block. Any module used by a program whether appearing in the program's globals or compiled separately, must be named in an import declaration. Modules, and the objects they export, always belong to the global scope of a program which uses them.

A module cannot be imported before it has been compiled, either as part of the importing program or by a previous invocation of the compiler. This prevents construction of mutually-referring modules. Access to separately compiled modules is discussed below.

Although a module declaration defines data and procedures which will become globals of any program importing the module, not everything declared in the module becomes known to the importer. A module specifies exactly what will be exported to the "outside world", and lists any other modules on which the module being declared is itself dependent.

The export declaration defines constants and types, declares variables, and gives the headings of procedures and functions whose complete specifications appear in the implement part of the module. It is exactly the items in the export declaration which become accessible to any other code which subsequently imports the module.

There need not be any procedures or functions in a module if its purpose is solely to declare types and variables for other modules.

Any constants, types and variables declared in the implement part will not be made known to importers of the module; they are only useful inside the module, and outside it they are hidden. Variables of the implement part of a module have the same lifetime as global program variables, even though they are hidden.

Any procedures or functions whose headings are exported by the module must subsequently be completely specified in its implement part. In this respect the headings in the export declaration are like FORWARD directives, and in fact the parameter list of such procedures need not be (but may be) repeated in the implement part. Procedures and functions which are not exported may be declared in the implement part; they are known and useful only within the module.

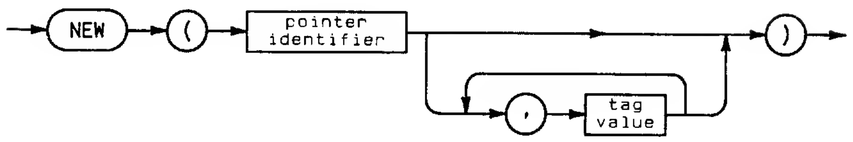
Separately compiled modules are called "library modules". To use library modules, a program imports them just as if they had appeared in the program block.

When an import declaration is seen, a module must be found matching each name in the import declaration. If a module of the required name appears in the compilation unit before the import declaration, the reference is to that module. Otherwise, external libraries must be searched.

The compiler option `$SEARCH 'string'$` names the order in which external libraries are searched. The parameter is a literal string describing the external libraries in an implementation-dependent fashion. Multiple files are specified by multiple strings. For instance, `$SEARCH 'file1','file2','file3'$`. This option may appear anywhere in a compilation unit, and overrides any previous SEARCH option.

new

This procedure allocates storage for a dynamic variable.



Item	Description/Default	Range Restrictions
pointer identifier	a pointer type variable	-
tag	case constant	-

Examples

```
new(ptr)
new(ptr,tag1,...,tagn)
```

Semantics

The procedure new(p) allocates storage for a dynamic variable on the heap and assigns its address to the pointer variable p. If insufficient heap space is available for the allocation, a run-time error occurs.

If the dynamic variable is a record with variants, then t may be used to specify a case constant. This constant only determines the amount of storage allocated. The procedure call does not actually assign it to the dynamic variable. For nested variants, you must list the values contiguously and in the order of their declaration.

If you call new for a record with variants and do not specify any case constants, the compiler determines storage by the size of the fixed part plus the size of the largest variant.

You should be careful when using an entire dynamic record variable allocated with one or more case constants as an operand in an expression, an actual parameter, or on the left side of an assignment statement. The variant may be smaller than the actual size at run time.

The pointer variable may be a component of a packed structure.

Pointer dereferencing accesses the actual values stored in a dynamic variable on the heap.

Example Code

```
PROGRAM show_new (output);
TYPE
  marital_status = (single, engaged, married, widowed, divorced);
  year = 1900..2100;
  ptr  = ^person_info;
  person_info = RECORD
                  name: string[25];
                  birdate: year;
                  next_person: ptr;
                  CASE status: marital_status OF
                    married..divorced: (when: year;
                                        CASE has_kids: boolean OF
                                          true: (how_many: 1..50)
                                        ););
                    engaged: (date: year)
                    single : ();
                END;


VAR
   p : ptr;
BEGIN                                 {Various legal calls of new.}
  .
  .
  new(p);
  .
  .
  new(p,engaged);
  .
  .
  new(p,married);
  .
  .
  new(p,widowed,false);
  .
  .
END.
```

NIL

This predefined constant is used when a pointer does not contain an address.

Semantics

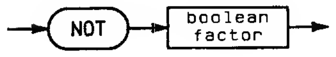
NIL is compatible with any pointer type. A NIL pointer (a pointer that has been assigned to NIL) does not point to any variable at all.

NIL pointers are useful in linked list applications where the "link" pointer points to the next element of the list. The last element's pointer can be assigned to NIL to indicate that there are no further elements in the list.

An error occurs when a NIL valued pointer is dereferenced.

NOT

This boolean operator complements a boolean factor.



Example

```
NOT done
```

Semantics

The NOT operator complements the value of the boolean factor following the NOT operator. The result is of type boolean.

Example Code

```
PROGRAM show_not(input,output);

VAR
  time, money : boolean;
  line        : string[255];
  test_file   : file;

BEGIN
  .
  .
  IF NOT (time AND money) THEN wait;
  .
  .
  WHILE NOT eof(test_file) DO
    BEGIN
      readln(test_file,line);
      writeln(line);
    END;
  .
  .
END.
```

Numbers

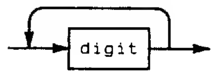
HP Pascal recognizes three sorts of numeric literals: integer, real, and longreal.

Integer Literals

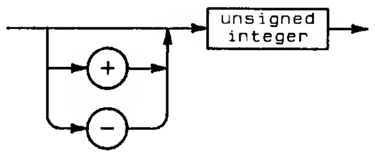
An integer literal consists of an sequence of digits from the subrange 0..9. No spaces may separate the digits for a single literal and leading zeroes are not significant. The compiler interprets unsigned integer literals as positive values.

The maximum unsigned integer literal is equal in value to the standard constant maxint. The minimum signed integer literal is equal in value to the standard constant minint. The actual value is implementation dependent.

Unsigned Integer:



Signed Integer

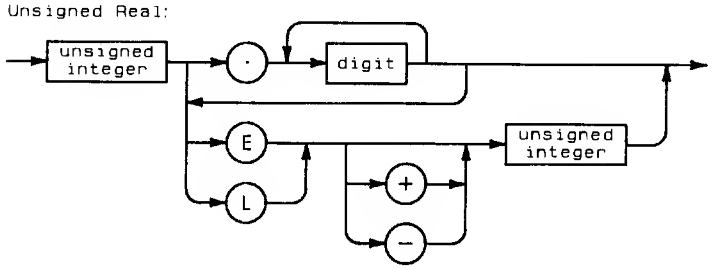


Real and Longreal Literals

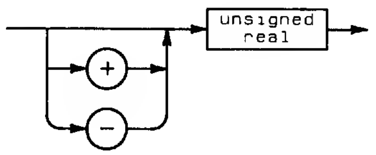
A real or longreal literal consists of a coefficient and a scale factor. An "E" preceding the scale factor is read as "times ten to the power of" and specifies a real literal. An "L" preceding the scale factor also means "times ten to the power of", but specifies a longreal literal.

Lowercase "e" and "l" are legal. At least one digit must precede and follow a decimal point. A number containing a decimal point and no scale factor is considered a real literal.

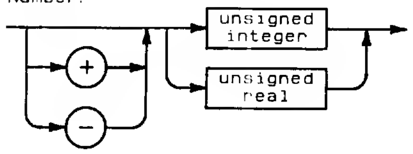
Unsigned Real:



Signed Real



Number:

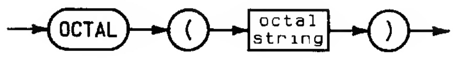


Examples

```
100                 {Integer.                     }
0.1                 {Real with no scale factor. }
5E-3                {Real with no decimal point.}
3.14159265358979L0  {Longreal.                    }
87.35e+8            {Real.                        }
```

octal

This function converts a string or PAC, whose literal value is an octal number, to an integer.



Item	Description/Default	Range Restrictions
octal string	string expression or PAC variable	implementation dependent

Examples

```
 Input                          Result
 octal(strng)
 octal('77')                     63
-octal('77')                    -63
```

If your particular implementation used 32-bit 2's complement notation, the following example would also work.

```
 octal('37777777701')           -63
```

Semantics

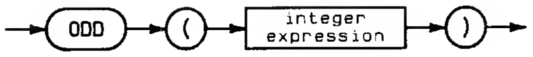
The function `octal`(s) converts s to an integer. S is interpreted as an octal value.

The three numeric conversion functions are `binary`, `hex`, and `octal`. All three accept arguments which are string or PAC variables, or string literals. The compiler ignores leading and trailing blanks in the argument. All other characters must be legal digits in the indicated base.

Since `binary`, `hex`, and `octal` return an integer value, all bits must be specified if a negative result is desired. Alternatively, you may negate the positive representation.

odd

This function returns `true` if the integer expression is odd, and `false` otherwise.



Examples

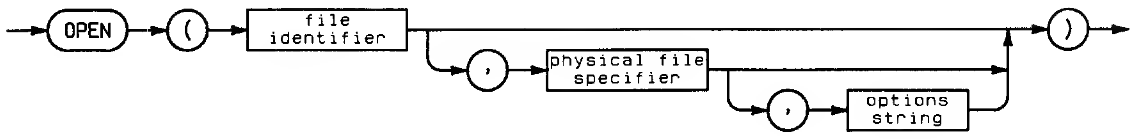
Input	Result
`odd(int_var)`	
`odd(ord(color))`	
`odd(2 + 4)`	`false`
`odd(-32767)`	`true`
`odd(32768)`	`false`
`odd(0)`	`false`

OF

See ARRAY, CASE, FILE, and String Constructor.

open

This procedure opens a file in the read-write state and places the current position at the beginning of the file.



Item	Description/Default	Range Restrictions
file identifier	name of a logical file	file cannot be of type text
physical file specifier	name to be associated with f; must be a string expression or PAC variable	-
options string	a string expression or PAC variable	implementation dependent

Examples

```
open(file_var)
open(file_var,phy_file_spec)
open(file_var,phy_file_spec,opt_str)
open(filvar,'TESTFILE')
```

Semantics

The procedure `open`(f) opens **f** in the read-write state and places the current position at the beginning of the file. The function `eof` returns `false`, unless the file is empty. The buffer variable **f^** is undefined.

After a call to `open`, **f** is said to be a direct access file. You may read or write data using the procedures `read`, `write`, `readdir`, or `writedir`. The procedure `seek` and the functions `lastpos` and `maxpos` are also legal. `Eof`(f) becomes `true` when the current position is greater than the highest-indexed component ever written to **f**.

Direct access files have a maximum number of components. The function `maxpos` returns this number. On implementations that allow direct access files to be extended, `maxpos` returns the value of `maxint` (the maximum possible number of components).

The `lastpos` function returns the index of the highest-written component of a direct access file.

You cannot open a textfile for direct access since its format is incompatible with direct access operations.

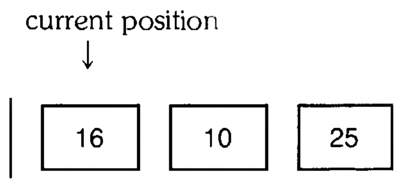
When a file is specified, the system will close any physical file previously associated with **f**.

When **f** does not appear as a program parameter and s is not specified, the system maintains any previous association of a physical file with **f**. If there is no such association, it opens a temporary nameless file. This file cannot be saved. It becomes inaccessible after the process terminates or the physical-to-logical file association changes.

Illustration

Suppose examp_file is a file of `integer` with three components. To perform both input and output, we call `open`:

```
open(examp_file);
```



state: read-write
examp_file^: undefined
eof(examp_file): false

Operators

An operator performs an action on one or more operands and produces a value.

An operand denotes an object which an operator acts on to produce a value. An operand may be a literal, a declared constant, a variable, a set constructor, a function call, a dereferenced pointer, or the value of another expression.

Operators are classified as arithmetic, boolean, set, relational, and concatenation operators. A particular symbol may occur in more than one class of operators. For example, the symbol "+" is an arithmetic, set and concatenation operator representing numeric addition, set union, and string concatenation, respectively.

Precedence ranking determines the order in which the compiler evaluates a sequence of operators (see Operator Precedence).

The value resulting from the action of an operator may in turn serve as an operand for another operator.

Arithmetic Operators

Arithmetic operators perform integer and real arithmetic. They include +, -, *, /, DIV, and MOD.

Most arithmetic operators permit real, longreal, integer, or integer subrange operands. DIV and MOD, however, only accept integer operands.

In general, the type of its operands determines the result type of an arithmetic operator. In certain cases, the compiler implicitly converts an operand to another type (see below).

Operator	Result
+ (unary)	The value of a single operand which may be any numeric type.
− (unary)	The negated value of a single operand which may be any numeric type.
+ (addition)	The sum of two operands which may be any but not necessarily the same numeric type.
− (subtraction)	The difference of two operands which may be any but not necessarily the same numeric type.
* (multiplication)	The product of two operands which may be any but not necessarily the same numeric type.
/ (division)	The quotient of two operands which may be any but not necessarily the same numeric type. If both operands are type integer or integer subrange, the result is, nevertheless, real.
DIV (division with truncation)	The truncated quotient of two operands which both must be type integer or integer subrange. The sign of the result is positive if the signs of the operands are the same, negative otherwise. The result is zero if the first operand is zero.
MOD (modulus)	The remainder when the right operand divides the left operand. Both operands must be integers or integer subrange, but an error occurs if the right operand is negative or zero. The result is always positive, regardless of the sign of the left operand, which must be parenthesized if it is a negative literal (see example). The result is zero if the left operand is zero. Formally, MOD is defined as i MOD j = i − ((i DIV j) * j) where i > 0 and j > 0. Or i MOD j = i − ((i DIV j) * j) + j where i < 0 and j > 0.

Implicit Conversion

The operators +, -, *, and / permit operands with different numeric types. For example, it is possible to add an integer and a real number. The compiler converts the integer to a real number and the result of the addition is real.

This implicit conversion of operands relies on a ranking of numeric types:

Rank	Type
highest	longreal
..	real
..	integer
lowest	integer subrange

If two operands associated with an operator are not the same rank, the compiler converts the operand of the lower rank to an operand of the higher rank prior to the operation. The result will have the type of the higher rank operand. In sum:

One operand type	Other operand type	Result type
integer-subrange	integer-subrange	integer
integer-subrange	integer	integer
integer	real	real
integer	longreal	longreal
real	longreal	longreal

Real division (/) is an exception. If both operands are integers or integer subranges, the compiler changes both to real numbers prior to the division and the result is real.

It is not legal to perform real or longreal arithmetic in a constant definition.

Examples

Expression	Result	
−(+10)	−10	{Unary −. }
5 + 2	7	{Addition with integer operands. }
5 − 2.0	3.0	{Subtraction with implicit conversion. }
5 *2	10	{Multiplication with integer operands. }
5.0 / 2.0	2.5	{Division with real operands. }
5 / 2	2.5	{Division with integer operands, real }
		{result. }
5.0L0 / 2	2.5L0	{Division with implicit conversion. }
5 DIV 2	+2	{Division with truncation. }
5 DIV (−2)	−2	
−5 DIV 2	−2	
−5 DIV (−2)	+2	
5 MOD 2	+1	{Modulus. }
5 MOD (−2)	error	{Right operand must be positive. }
(−5) MOD 2	+1	{Result is positive regardless of }
		{sign of left operand, which is }
		{parenthesized since MOD has higher }
		{precedence than −. }
		{See Operator Precedence }

Boolean Operators

The boolean operators perform logical functions on boolean type operands and produce boolean results. The boolean operators are NOT, AND, and OR.

When both operands are boolean, = denotes equivalence, <= implication, and <> exclusive or.

Operator	Result
NOT (logical negation)	The logical negation of a single boolean operand according to the following table:

a	NOT a
true	false
false	true

Operator	Result
AND (logical and)	The evaluation of two boolean operands produces a boolean result according to the following table:

a	b	a AND b
true	true	true
true	false	false
false	true	false
false	false	false

Operator	Result
OR (inclusive or)	The evaluation of two boolean operands produces a boolean result according to the following table:

a	b	a OR b
true	true	true
true	false	true
false	true	true
false	false	false

The compiler can be directed to perform partial evaluation of boolean operators used in statements. For instance:

```
   IF right_time AND right_place THEN ...
```

By specifying the $PARTIAL_EVAL ON$ compiler directive, if "right_time" is `false`, the remaining operators will not be evaluated since execution of the statement depends on the logical AND of both operators. (Both operators would have to be `true` for the logical AND of the operators to be `true`.)

Similarly, the logical OR of two operators would be `true` even if only one of the operators was `true`.

With careful planning of "most likely" values for boolean operators, partial evaluation can reduce execution time of a program.

Example Code

```
IF NOT possible THEN forget_it;

WHILE time AND money DO your_thing;

REPEAT...UNTIL tired OR bored;

IF has_rope = true DO skip;

IF pain <= heartache THEN try_it;

FUNCTION NAND (A, B : BOOLEAN) : BOOLEAN;
     NAND := NOT(A AND B);  {NOT AND}

FUNCTION XOR (A, B : BOOLEAN) : BOOLEAN;
      XOR := NOT(A AND B) AND (A OR B);  {EXCLUSIVE OR}

FUNCTION XOR (A, B : BOOLEAN) : BOOLEAN;
      XOR := A <> B;
```

Concatenation Operators

The concatenation operator + concatenates two operands. The operands may be string variables, string literals, function results of type `string`, or some combination of these types.

If one of the operands is type `string`, the result of the concatenation is also type `string`. If both operands are string literals, the result is a string.

It is not legal to use the concatenation operator in a constant definition.

Example Code

```
VAR
   s1,s2: string[80];
BEGIN
    .
   s1:= 'abc';
   s2:= 'def';
   s1:= s1 + s2;     {S1 is now 'abcdef'}
    .
   s2:= 'The first six letters are ' + s1;
    .
END.
```

Relational Operators

Relational operators compare two operands and return a boolean result. The relational operators are <, <=, =, <>, >=, >, and IN. The < operator means "less than"; <= "less than or equal"; = "equal"; <> "not equal"; >= "greater than or equal"; > "greater than"; and IN indicates set membership.

Depending on the type of their operands, relational operators are classified as simple, set, pointer, or string relational operators.

Simple Relational Operators

A simple relational operator has operands of any simple type, i.e. `integer`, `boolean`, `char`, `real`, `longreal`, enumerated, or subrange. All the operators listed above except IN may be simple relational operators. The operands must be type compatible, but the compiler may implicitly convert numeric types before evaluation (see Arithmetic Operators).

For numeric operands, simple relational operators impose the ordinary definition of ordering. For char operands, the ASCII collating sequence defines the ordering. For enumerated operands, the sequence in which the constant identifiers appear in the type definition defines the ordering. Thus the predefinition of `boolean` as

```
TYPE boolean = (false, true);
```

means that `false` < `true`.

If both operands are boolean, the operator = denotes equivalence, <= implication, and <> exclusive or.

Set Relational Operators

A set relational operator has set operands. The set relational operators are =, <>, >=, <=, and IN.

The operators = and <> compare two sets for equality or inequality, respectively. The <= operator denotes the subset operation, while >= indicates the superset operation. Set A is a subset of Set B if every element of A is also a member of B. When this is true, B is said to be the superset of A.

The IN operator determines if the left operand is a member of the set specified by the right operand. When the right operand has the type SET OF T, the left operand must be type compatible with T. To test the negative of the IN operator, use the following form:

```
NOT (element IN set)
```

Pointer Relational Operators

You can use the operators = and <> to compare two pointer variables for equality or inequality, respectively. Two pointer variables are equal only if they point to exactly the same object on the heap. You may compare two pointers of the same type or the constant NIL to a pointer of any type.

String Relational Operators

You may use the string relational operators =, <>, <, <=, >, or >= to compare operands of type `string`, PAC, `char`, or string literals.

The system performs the comparison character by character using the order defined by the ASCII collating sequence.

If one operand is a string variable, the other operand may be a string variable or string literal. If the operands are not the same length and the two are equal up to the length of the shorter, the shorter operand is less. For example, if the current value of S1 is "abc" and the current value of S2 is "ab", then S1 > S2 is `true`. It is not possible to compare a string variable with a PAC or char variable.

If one operand is a PAC variable, the other may be a PAC (of any length) or a string literal no longer than the first operand. If shorter, the string literal is blank filled prior to comparison. It is not possible to compare a PAC with a string or char variable.

If one operand is a char variable, the other may be a char variable or a single-character string literal. It is not possible to compare a char variable with a string or PAC variable.

If one operand is a string literal, the other may be a string variable, a PAC, a string literal, or a char variable provided that the string literal is only of length 1.

The following table summarizes these rules. The standard function `strmax`(s) returns the maximum length of the string variable s. The standard function `strlen`(s) returns the current length of the string expression **s**.

A string constant is considered a string literal when it appears on either side of a relational operator.

String, PAC, Char, String Literal Comparison

A/<relop>/B	**string**	**PAC**	**char**	**string literal**
string	Length of comparison based on smaller strlen	Not allowed	Not allowed	Length of comparison based on smaller strlen
PAC	Not allowed	The shorter of the two is padded with blanks	Not allowed	Only if A length >= strlen(B) B is blank filled if necessary
char	Not allowed	Not allowed	Yes	Only if strlen(B) = 1
string literal	Length of comparison based on smaller strlen	The shorter of the two is padded with blanks	Only if strlen(A) = 1	Yes A or B is blank filled if necessary

Example Code

```
PROGRAM show_relational;
TYPE
  color = (red, yellow, blue);
VAR
  a,b,c: boolean;
  p,q: ^boolean;
  s,t: SET OF color;
  col: color;
  stg: string[10];
BEGIN
   .
   .                         {Types of relational operators:    }
  b := 5 > 2;                                    {simple,          }
  b := 5 < 25.0L+1;                              {simple,          }
  b := a AND (b OR (NOT c AND (b <= a)));        {implication,}
  IF (p = q) AND (p <> NIL) THEN p^:= a = b;     {pointer,         }
  b := s <> t;                                   {set,             }
  b := s <= t;                                   {set, subset,}
  b := col IN [yellow, blue];                    {set, IN,         }
  stg := 'alpha';
  c := stg > 'beta';                             {string,          }
END.
```

SET Operators

The set operators perform set operations on two set operands. The result is a third set. The set operators are +, –, and *.

Operator	Result
+ (union)	A set whose members are all the elements present in the left set operand and those in the right, including members present in both sets.
– (difference)	A set whose members are the elements which are members of the left set but are not members of the right set.
* (intersection)	A set whose members are only those elements present in both of the set operands.

Operands used with set operators may be variables, constant identifiers, or set constructors. The base types of the set operands must be type compatible with each other.

Example Code

```
PROGRAM show_setops;
VAR
  a, b, c: SET OF 1..10;
  x : 1..10;
BEGIN
  .
  .
  a:= [1, 3, 5];
  b:= [2, 4];
  c:= [1..10];
  x:= 9;
  a:= a + b          {Union; a is now [1, 2, 3, 4, 5].       }
  b:= c - a          {Difference; b is now [6, 7, 8, 9, 10].}
  c:= a * b          {Intersection; c is now [].            }
  c:= [2, 5] + [x]    {Set constructor operands; c is now   }
END.                  {[2, 5, 9].                           }
```

Operator Precedence

The precedence ranking of a HP Pascal operator determines the order of its evaluation in an unparenthesized sequence of operators. The four levels of ranking are:

Precedence	Operators
highest	NOT
.	*, /, DIV, MOD, AND
.	+, −, OR
lowest	<, <=, <>, =, >=, >

The compiler evaluates higher precedence operators first. For example, since * ranks above +, it evaluates these expressions identically:

```
(x + y * z)    and    (x + (y * z))
```

When a sequence of operators has equal precedence, the order of evaluation is implementation dependent.

If an operator is commutative (e.g. *), the compiler may choose to evaluate the operands in any order.

Within a parenthesized expression, of course, the compiler evaluates the operators and operands without regard for any operators outside the parentheses.

Summary

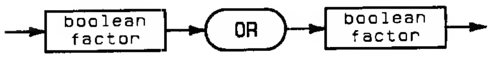
The following table lists each HP Pascal operator together with its actions, permissible operands, and type of results. In the table, the term "real" indicates both `real` and `longreal` types.

HP Pascal Operators

Operator	Actions	Type of Operands	Type of Results
+	addition set union concatenation	real, integer any set type T string, string lit.	real, integer T string
−	subtraction set difference	real, integer any set type T	real, integer T
*	multiplication set intersection	real, integer any set type T	real, integer T
/	division	real, integer	real
DIV	division with truncation	integer	integer
MOD	modulus	integer	integer
AND	logical 'and'	boolean	boolean
OR	logical 'or'	boolean	boolean
NOT	logical negation	boolean	boolean
<	less than	any simple type string or PAC	boolean boolean
>	greater than	any simple type string or PAC	boolean boolean
<=	less than or equal, set subset	any simple type string or PAC any set	boolean boolean boolean
>=	greater than or equal, set superset	any simple type string or PAC any set	boolean boolean boolean
=	equal to	any simple type string or PAC any set type pointer	boolean boolean boolean boolean
<>	not equal to	any simple type string or PAC any set type pointer	boolean boolean boolean boolean
IN	set membership	left operand: any ordinal type T right operand: set of T	boolean

OR

This boolean operator returns the logical OR of its two factors.



Example

```
ok OR quit
```

Semantics

The truth table is:

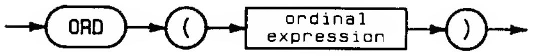
A	B	A OR B
false	false	false
false	true	true
true	false	true
true	true	true

Example Code

```
PROGRAM show_or(input,output);

  VAR
    ch     : char;
    time   : boolean;
    energy : boolean;

 BEGIN
   .
   .
   IF time OR energy then doit;
   .
   .
   IF (ch = 'Y') OR (ch = 'y') THEN ch := 'Y';
   .
   .
END.
```

ord

This standard function returns an integer designating the position of the argument in an ordered set.



Examples

```
Input                          Result
ord(ord_exp)
ord('a')                        97
ord('A')                        65
ord(-1)                         -1
ord(yellow)                     2  {TYPE color=(red,blue,yellow)}
ord(red)                        0
```

Semantics

The function `ord(x)` returns the integer representing the ordinal associated with the value of x. If x is an integer, x itself is returned. If x is type `char`, the result is an integer value between 0 and 255 determined by the ASCII order sequence. If x is any other ordinal type (i.e., a predefined or user-defined enumerated type), then the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers starting at zero. For example, since the standard type `boolean` is predefined as:

```
TYPE boolean = (false,true)
```

the call `ord(false)` returns 0, and the call `ord(true)` returns 1.

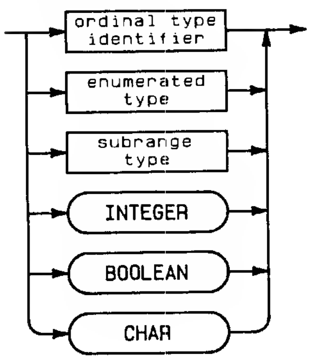
For any character ch, the following is true:

```
chr(ord(ch)) = ch
```

Ordinal Types

Ordinal types are types that can be uniquely mapped into the set of natural numbers.

Ordinal Type:



Ordinal types include enumerated types, subrange types, integers, booleans, and characters (char type).

Ordinal types are declared by enumerating all of the possible values that their variables and functions can possess. Predefined ordinal types include integers, boolean values, and characters.

Permissible Operators
Any of the relational operators may be used with ordinal types. The IN (membership test) operator may also be used with ordinal types.

For relational tests, the two factors must be of the same type. When membership tests are performed, the left-operand type must be a single ordinal value while the right-operand is of a SET type.

Permissible Functions
The following functions may be used with all ordinal types.

succ	This function returns the next value in the list of possible values the variable may possess. The succ of the last value is undefined.
pred	This function returns the previous value in the list of possible values. The pred of the first value is undefined.
ord	This function returns the ordinal number of the given value.

OTHERWISE

In HP Pascal, a CASE statement may include an OTHERWISE part.

See CASE.

output

The standard textfiles `input` and `output` often appear as program parameters. When they do, there are several important consequences:

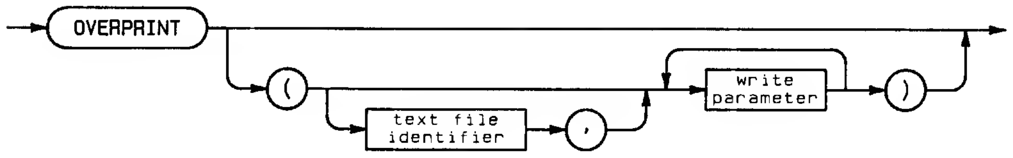
1. You may not declare `input` and `output` in the source code.
2. The system automatically resets `input` and rewrites `output`.
3. The system automatically associates `input` and `output` with the implementation dependent physical files.
4. If certain file operations omit the logical file name parameter, `input` or `output` is the default file. For example, the call `read`(x), where x is some variable, reads a value from `input` into x. Or consider:

```
PROGRAM sample (output);
BEGIN
  writeln('I like Pascal!');
END.
```

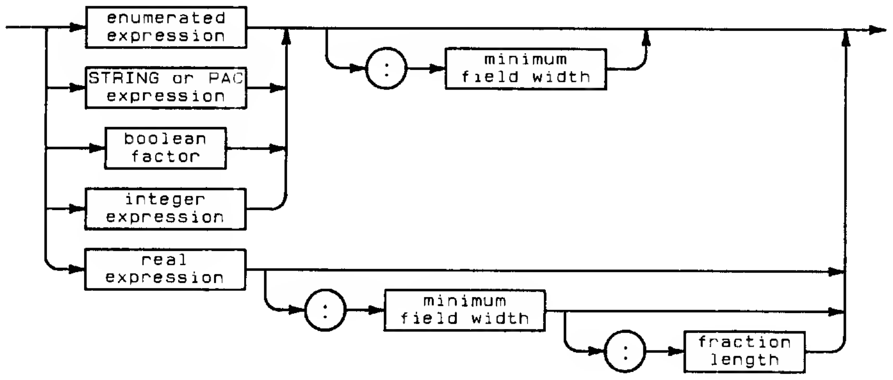
The program displays the string literal on the terminal screen. `Output` must appear as a program parameter; `input` need not appear, however, since the program doesn't use it.

overprint

This procedure writes a special character to a textfile which suppresses the generation of a line-feed after the item is printed.



Write Parameter



Item	Description/Default	Range Restrictions
textfile identifier	variable of type text default = `output`	file must be opened
write parameter	see drawing	-
minimum field width	integer expression	greater than 0
fraction length	integer expression	greater than 0

Examples

```
overprint(file_var)
overprint(file_var,exp)
overprint(file_var,exp1,...,expn)
overprint(exp)
overprint(exp1,...,expn)
overprint
```

Semantics

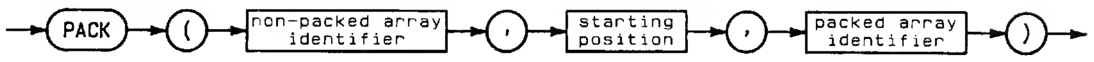
The procedure `overprint`(f) writes a special line marker on the textfile f and advances the current position. When f is printed, this special marker causes a carriage return but suppresses the line feed. This means the printer will print the line after the special marker over the line preceding it.

After the execution of `overprint`(f), the buffer variable f^ is undefined and `eoln`(f) is false.

The expression parameter behaves exactly like the equivalent parameter for the procedure `write`.

pack

This standard procedure transfers data from unpacked arrays to packed arrays.



Item	Description/Default	Range Restrictions
non-packed array identifier	variable of type array	see semantics
starting position	expression which is type compatible with the index of the non-packed array	-
packed array identifier	variable of type PACKED array	see semantics

Example

```
pack(array,start_pos,packed_array)
```

Semantics

Assuming `a: ARRAY[m..n] OF t` and `x: PACKED ARRAY [u..v] OF t`; the procedure `pack`(a,i,z) assigns components of the unpacked array a, starting at component i, to each component of the packed array z. The unpacked array must be as long as or longer than the packed array, i.e. $n-m >= v-u$. The value of i must be greater than or equal to m, the lower bound of a. Since all the components of z are assigned a value, the normalized value of i must be less than or or equal to the difference between the lengths of a and z plus 1, i.e. $i-m+1 <= (n-m) - (v-u) + 1$. Otherwise, an error occurs when `pack` attempts to access a non-existent component of a (see example below).

The component types of arrays a and z must be type identical. The index types of a and z, however, may be incompatible.

The call `pack`(a,i,z) is equivalent to:

```
BEGIN
  k:= i;
  FOR j:= u TO v DO
    BEGIN
     z[j]:= a[k];
      IF j <> v THEN k:= succ(k);
    END;
END;
```

where k and j are variables that are type compatible with the index type of a and the index type of z, respectively.

Example Code

```
PROGRAM show_pack (input,output);
TYPE
  clothes = (hat, glove, shirt, tie, sock);
VAR
  dis : ARRAY [1..10] OF clothes;
  box : PACKED ARRAY [1..5] of clothes;
  index: integer;
  .
  .
BEGIN
  .
  .
  index:= 1;
  pack(dis,index,box);    {After pack executes, box contains    }
  .                       {the first 5 components of dis.       }
  .
  index:= 8;
  pack(dis,index,box);    {An error results when pack attempts  }
  .                       {to access non-existent 11th component}
  .                       {of dis.                              }
 END.
```

PACKED

This reserved word indicates that the compiler should optimize data storage.

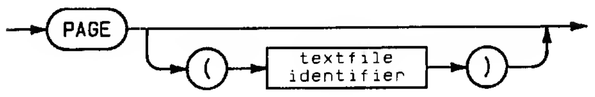
PACKED may appear with an ARRAY, RECORD, SET, or FILE.

By declaring a PACKED item, the amount of memory needed to store an item is generally reduced.

Whether data storage is optimized depends on the implementation.

page

This procedure writes a special character to a textfile which causes the printer to skip to the top of form when the file is printed.



Item	Description/Default	Range Restrictions
textfile identifier	variable of type text; default = `output`	file must be open

Examples

```
page(text_file)
page
```

Semantics

The procedure `page`(f) writes a special character to the textfile f which causes the printer to skip to the top of form when f is printed. The current position in f advances and the buffer variable f^ is undefined.

If f is omitted, the system uses the standard file `output`.

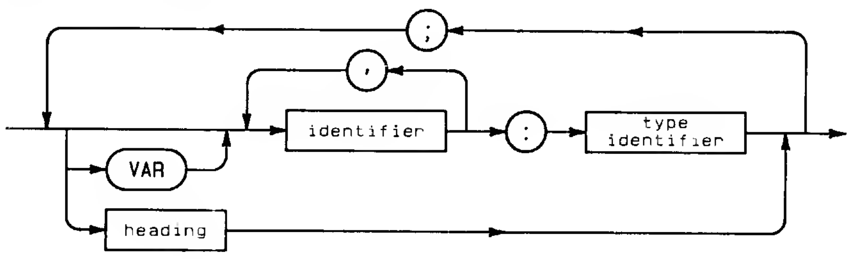
Parameters

A procedure or function is declared, the heading may optionally include a list of parameters. This list is called the formal parameter list.

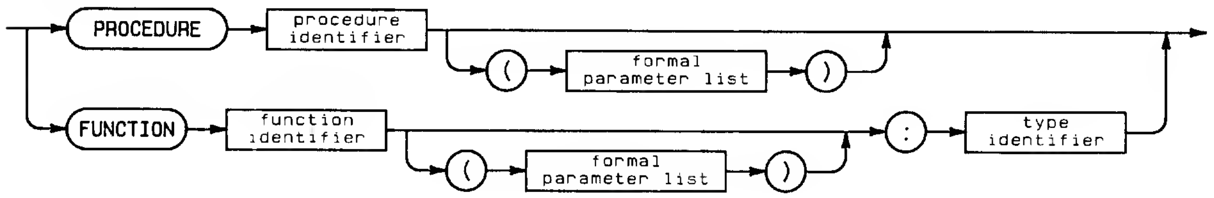
A procedure statement or function call in the body of a block provides the matching actual parameters which correspond by their order in the list. The list of actual parameters must be assignment compatible with their corresponding formal parameters.

The four sorts of formal parameters are value, variable, function, and procedure parameters. Value parameters are identifiers followed by a colon (:) and a type identifier. Variable parameters are identical with value parameters except they are preceded by the reserved word VAR. Function or procedure parameters are function or procedure headings.

Formal Parameter List



Heading:



You may repeat and intermix the four types of formal parameters. Several identifiers may appear separated by commas. These identifiers will then represent formal variable or value parameters of the same type.

A formal value parameter functions as a local variable during execution of the procedure or function. It receives its initial value from the matching actual parameter. Execution of the procedure or function doesn't affect the actual parameter, which, therefore, may be an expression.

A formal variable parameter represents the actual parameter during execution of the procedure. Any changes in the value of the formal variable parameter will alter the value of the actual parameter, which, therefore, must be a variable. A `string` type formal variable parameter need not specify a maximum length, it will assume the type of the actual parameter.

A formal procedure or function parameter is a synonym for the actual procedure or function parameter. The parameter lists, if any, of the actual and formal procedure or function parameters must be congruent.

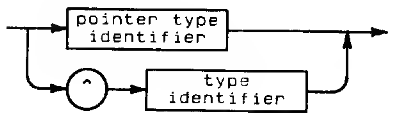
Example Code

```
PROGRAM show_formparm;
 VAR
  test: boolean;

FUNCTION chek1 (x, y, z: real): boolean;
  BEGIN
    {Perform some type of validity check on x, y, z }
    {and return appropriate value.                  }
   END;

FUNCTION chek2 (x, y, z: real): boolean;
  BEGIN
    {Perform an alternate validity check on x, y, z }
    {and return appropriate value.                  }
  END;

PROCEDURE read_data (FUNCTION check (a, b, c: real): boolean);
  VAR p, q, r: real;
  BEGIN
    {read and validate data}
    readln (p, q, r);
    IF check (p, q, r) THEN ...
  END;

BEGIN {show_formparm}
   .
  IF test THEN read_data (chek1)
    ELSE read_data (chek2);
   .
END.


PROGRAM show_varparm(output);

VAR
  i,j : integer;

PROCEDURE fix(VAR i : integer; j : integer);

BEGIN
  i := j;  {i is passed by reference, it will return equal to 42}
  j := 0;  {j is passed by value, this assignment will }
           {not change the value of j in the main program}
 END;

BEGIN {show_varparm}
  i:= 0;
   j:= 42;
  fix(i,j);
  IF i = j THEN writeln('They both = 42');
END.
```

Pointers

A pointer references a dynamically allocated variable on the heap. A pointer type consists of the caret (^) and a type identifier.

Pointer Type:



The type may be any type, including file types. The @ symbol may replace the caret.

You need not have previously defined the type appearing after the caret. This is an exception to the general rule that Pascal identifiers are first defined and then used. However, you must define the identifier after the caret within the same declaration part, although not necessarily within the same TYPE section.

A type identifier used in a pointer type declaration in an EXPORT section need not be defined until the IMPLEMENT section.

The pointer value NIL belongs to every pointer type; it points to no variable on the heap.

Permissible Operators

assignment:	`:=`
equality:	`=, <>`

Standard Procedures

pointer parameters:	`new, dispose, mark, release`

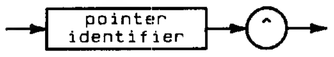
Examples

```
TYPE
  ptr1   = ^rec1;
  ptr2   = ^rec2;
  rec1   = RECORD
             f1, f2: integer;
             link:   ptr2;
           END;
  rec2   = RECORD
             f1, f2: real;
             link:   ptr1;
           END;
```

Pointer dereferencing

A pointer variable points to a dynamically-allocated variable on the heap. The current value of this variable may be accessed by dereferencing its pointer.

Pointer dereferencing occurs when the caret symbol (^) appears after a pointer designator in source code.



The pointer designator may be the name of a pointer or selected component of a structured variable which is a pointer. The @ symbol may replace the caret.

If the pointer is NIL or undefined, dereferencing causes an error.

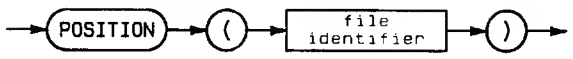
A dereferenced pointer can be an operand in an expression.

Examples

```
PROGRAM show_pointerderef (output);
TYPE
  p = ^integer;
VAR
  a,b      : integer;
  p_array  : ARRAY [1..10] OF p;
  ptr      : p;
 BEGIN
   .
  p_array[a]^:= a + b;
   .
  writeln(ptr^ * 2);         {Dereferenced pointer is operand. }
   .
END.
```

position

This function returns the index of the current file component.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	must not be a textfile

Example

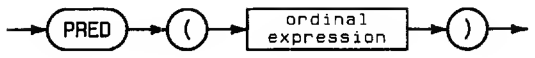
```
position(file_var)
```

Semantics

The function `position`(f) returns the integer index of the current component of f, starting from 1. Input or output operations will reference this component. **f** must not be a textfile. If the buffer variable f^ is full, the result is the index of the component in the buffer.

pred

This function returns the value whose ordinal number is one less than the ordinal number of the argument.



Examples

Input	Result
`pred(ord_var)`	
`pred(1)`	`0`
`pred(-5)`	`-6`
`pred('B')`	`'A'`
`pred(true)`	`false`

Semantics

The function `pred`(x) returns the value, if any, whose ordinal number is one less than the ordinal number of x. The type of the result is identical with the type of x. A run-time error occurs if `pred`(x) does not exist. For example, suppose:

```
TYPE day = (monday,tuesday,wednesday)
```

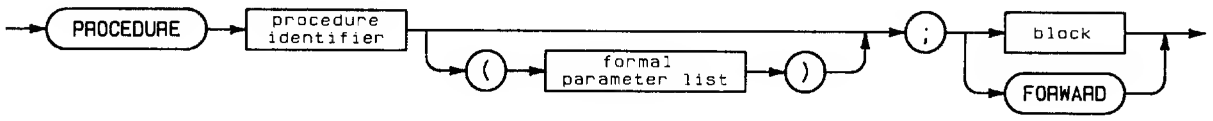
Then,

```
pred(tuesday) = monday
```

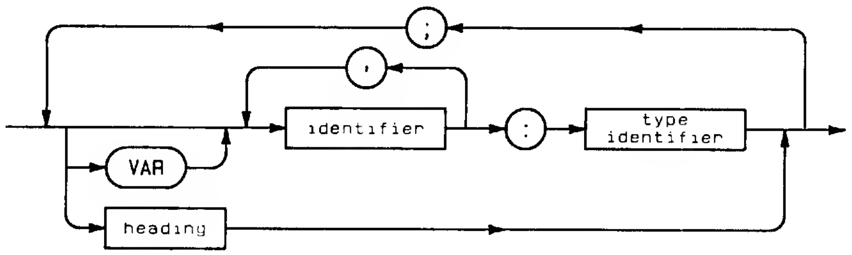
but `pred(monday)` is undefined.

PROCEDURE

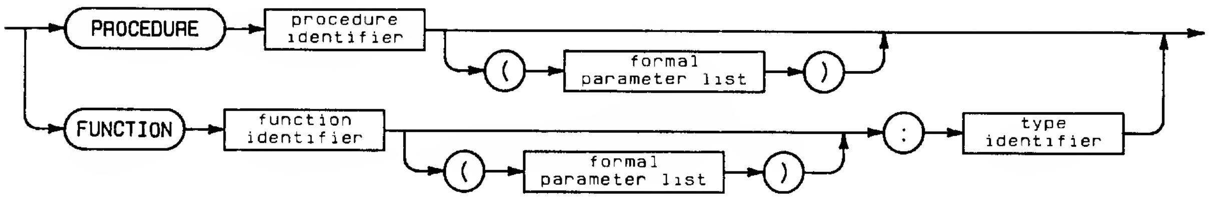
A procedure is a block which is activated with a PROCEDURE statement. A procedure declaration consists of a procedure heading, a semi-colon (;), and a block or a directive followed by a semi-colon.



Formal Parameter List



Heading:



Item	Description/Default	Range Restrictions
procedure identifier	name of a user-defined procedure	any valid identifier
formal parameter list	see diagram	-
heading	see drawing	-

Semantics

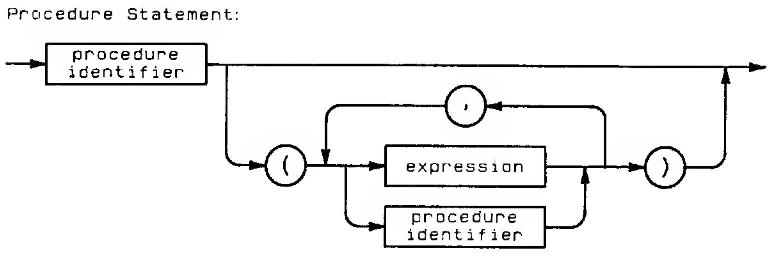
The procedure heading consists of the reserved word PROCEDURE, an identifier (the procedure name), and, optionally, a formal parameter list.

A directive can replace the procedure block to inform the compiler of the location of the block. A procedure block consists of an optional declaration part and a compound statement.

Procedure declarations must occur at the end of a declaration part after label, constant, type, and variable declarations and after the module declarations in the outer block. You can intermix procedure and function declarations.

Procedures

A procedure statement transfers program control to the block of a declared or standard procedure. After the procedure has executed, control is returned to the statement following the procedure call. A procedure statement consists of a procedure identifier and, if required, a list of actual parameters in parentheses.



The procedure identifier must be the name of a standard procedure or a procedure declared in a previous procedure declaration.

The declaration may be an actual declaration (i.e. heading plus body), a forward declaration, or it may be the declaration of a procedure parameter.

If a procedure declaration includes a formal parameter list, the procedure statement must supply the actual parameters. The actual parameters must match the formal parameters in number, type and order. There are four kinds of parameters: value, variable, procedure and function.

Actual value parameters are expressions which must be assignment compatible with the formal value parameters.

Actual variable parameters are variables which must be type identical with the formal variable parameters. Components of a packed structure cannot appear as actual variable parameters.

Actual procedure or function parameters are the names of procedures or functions declared in the program. Standard procedures or functions are not legal actual parameters.

If a procedure or function passed as an actual parameter accesses any entity non-locally upon activation, then the entity accessed is one which was accessible to the procedure or function when it was passed as a parameter. For example, suppose Procedure A uses the non-local variable x. If A is then passed as an actual procedure parameter to Procedure B, it will still be able to use x, even if x is not otherwise accessible from B.

The formal parameters, if any, of an actual procedure or function parameter must be congruent with the formal parameters of the formal procedure or function parameter. Two formal parameter lists are congruent if they contain an equal number of parameters and the parameters in corresponding positions are equivalent. Two parameters are equivalent if any of the following conditions are true.

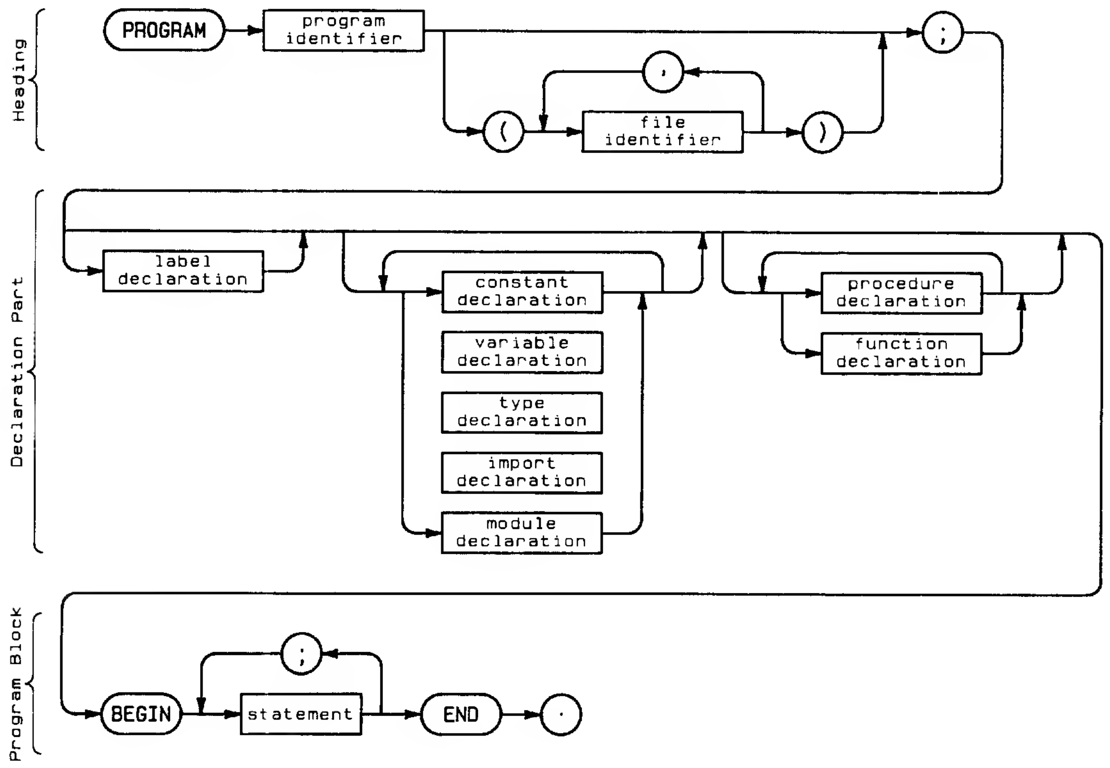
1. They are both value parameters of the identical type. Assignment compatibility is not sufficient.
2. They are both variable parameters of the identical type.
3. They are both procedure parameters with congruent parameter lists.
4. They are both function parameters with congruent parameter lists and identical result types.

Example Code

```
PROGRAM show_pstate (output);

  PROCEDURE wow; forward;         {Forward declaration.      }

  PROCEDURE bow;
    BEGIN
      write('bow-');
      wow;                        {procedure used before     }
    END;                          { it is defined            }

  PROCEDURE wow;                  {Forward procedure defined}
    BEGIN
      write('wow');
    END;

  PROCEDURE actual_proc           {Actual procedure declaration.}
     (a1: integer;
      a2: real);
     BEGIN
       IF a2 < a1 THEN
          actual_proc (a1, a2-a1)  {recursive call}
        ...
    END;

  PROCEDURE outer                  {Another actual declaration. }
     (a: integer;
      PROCEDURE proc_parm
      (p1: integer; p2 : real));

    PROCEDURE inner;   {nested procedure}
      BEGIN
        actual_proc (50, 50.0);
      END;

    BEGIN {outer}               {Calling a                    }
       writeln ('Hi');          {predefined procedure,        }
       inner;                   {inner procedure,             }
       proc_parm (2, 4.0);      {procedure parameter.         }
    END;  {outer}

BEGIN {show_pstate}
  outer (30, actual_proc);        {procedure parameters.        }
END.  {show_pstate}
```

PROGRAM

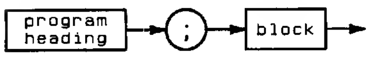
An HP Pascal program consists of three major parts; the program heading, the program declaration, and the program block.



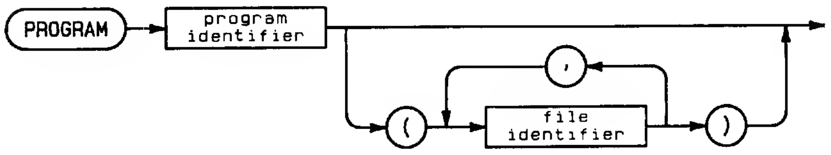
See Programs.

Programs

An HP Pascal compiler will successfully compile source code which conforms to the syntax and semantics of an HP Pascal program. The form of an HP Pascal program consists of a program heading, a semicolon (;), a program block, and a period.



The program heading consists of the reserved word PROGRAM, an identifier (the program name) and an optional parameter list.

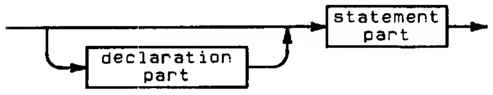


The identifiers in the parameter list are variables which must be declared in the outer block, except for the standard textfiles `input` and `output`.

`Input` and `output` are standard file variables which the system associates by default with system dependent files and devices which it opens automatically at the beginning of program execution. In HP Pascal, `input` or `output` need only appear as program parameters if some file operation, e.g. `read` or `write`, refers to them explicitly or by default.

Program parameters are often the names of file variables, but a logical file, i.e. a file declared in the program, need not necessarily appear as a program parameter. What must appear is system dependent.

The program block consists of an optional declaration part and a required statement part.



The declaration part (see next page) consists of definitions of labels, constants and types, and declarations of variables, procedures, functions, and modules. The statement part is made up of a compound statement which may be empty or may contain several simple or structured statements (see Statements). The statement part is also termed the "body" or "executable portion" of the block.

Example Code

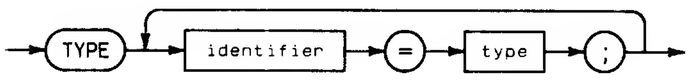
```
PROGRAM minimum;             {The minimum program the HP Pascal    }
BEGIN                        {compiler will process successfully: }
END.                         {no program parameters. }


PROGRAM show_form1 (output); {Uses the standard textfile output   }
 BEGIN                        {and the standard procedure writeln.}
  writeln ('Greetings!')
END.


PROGRAM show_form2 (input,output);
VAR
  a,b,total: integer;

FUNCTION sum (i,j: integer): integer;   {Function declaration    }
  BEGIN
    sum:= i + j
  END;

BEGIN
  write ('Enter two integers: ');
  prompt;
  readln (a,b);
  total:= sum (a,b);
  writeln ('The total is: ', total)
END.
```

Declaration Part

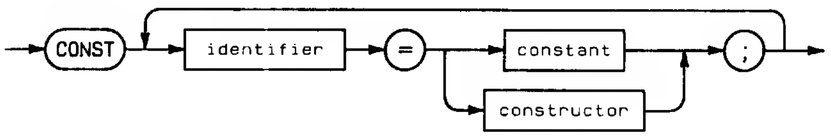
The declaration part of an HP Pascal program block defines the labels, declared constants, data types, variables, procedures, functions, and modules which will be used in the executable statements in the body of the block.

The reserved word LABEL precedes the declaration of labels; CONST or TYPE the definition of declared constants or types; VAR the declaration of variables; IMPORT a list of modules; MODULE the declaration of a module; PROCEDURE or FUNCTION the declaration of a procedure or a function.

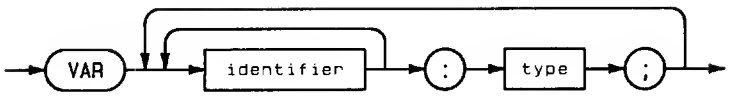
Type Declaration



Constant Declaration



Variable Declaration



Within a declaration part, label declarations must come first; procedure or function declarations last. You can intermix and repeat CONST and TYPE definition sections, VAR declaration sections (see example below) and MODULE declarations.

ANSI Standard Pascal does not allow any of the reserved words, LABEL, CONST, TYPE, or VAR to be used more than once.

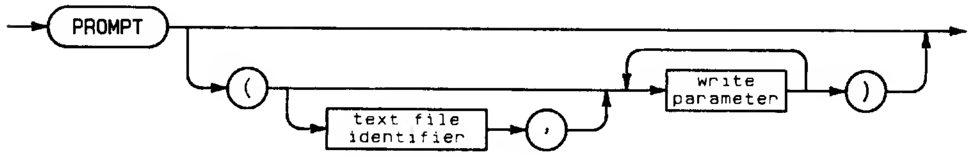
You can redeclare or redefine a standard declared constant, type, variable, procedure or function in a declaration part. You will, of course, lose any previous definition associated with that item.

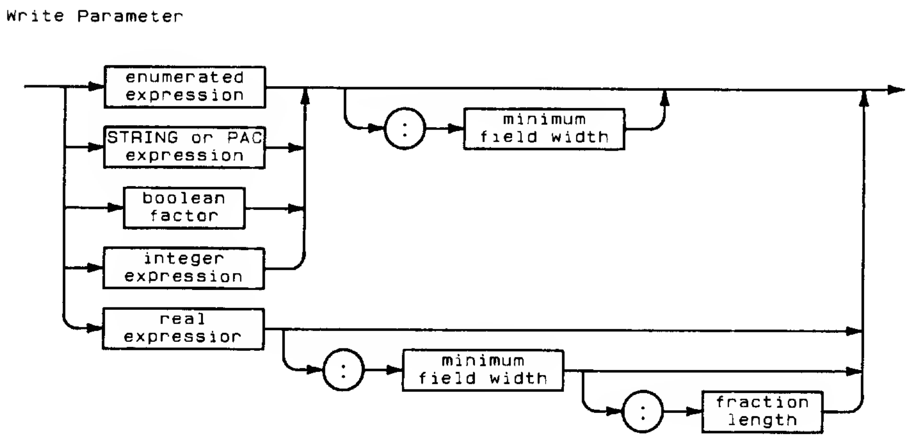
Example Code

```
PROGRAM show_declarepart;
LABEL  25;
VAR
  birthday: integer;
TYPE
  friends = (Joe, Simon, Leslie, Jill);
CONST
  maxnuminvitee = 3;
VAR
  invitee: friends;
PROCEDURE hello;
  BEGIN
    writeln('Hi');
  END;                              {End of declaration part.}

BEGIN                               {Beginning of body.      }
  .
  .
END.
```

prompt

This procedure causes the system to write any buffers associated with a textfile to the output device.





Item	Description/Default	Range Restrictions
textfile identifier	variable of type text; default = `output`	file must be opened to write
write parameter	see drawing	-
minimum field width	integer expression	greater than 0
fraction length	integer expression	greater than 0

Examples

```
prompt(file_var)
prompt(file_var,exp)
prompt(file_var,exp1,...,expn)
prompt(exp)
prompt(exp1,...,expn)
prompt
```

Semantics

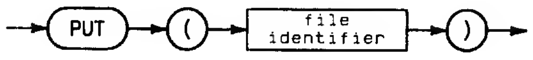
The procedure `prompt`(f) causes the system to write any buffers associated with textfile f to the device. `Prompt` does not write a line marker on f. The current position is not advanced and the buffer variable f^ becomes undefined.

You normally use `prompt` when directing I/O to and from a terminal. `Prompt` causes the cursor to remain on the same line after output to the screen is complete. The user may then respond with input on the same line.

The expression parameter e behaves exactly like the equivalent parameters in the procedure `write`.

put

This procedure assigns the value of the buffer variable to the current file component.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	file must be open to write

Example

```
put(file_var)
```

Semantics

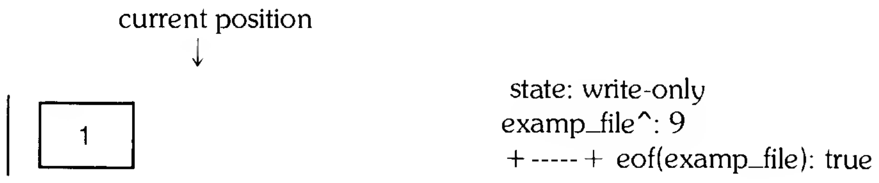
The procedure put(f) assigns the value of the buffer variable f^ to the current component and advances the current position. Following the call, f^ is undefined.

An error occurs if f is open in the read-only state.

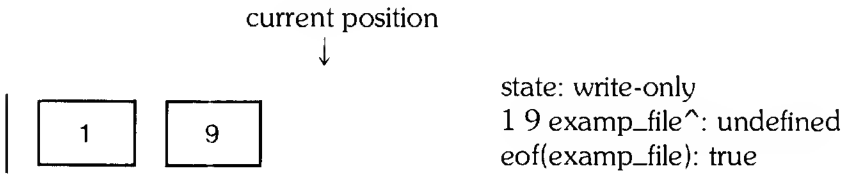
Illustration

Suppose examp_file is a file of integer with a single component opened in the write-only state by append. Furthermore, we have assigned 9 to the buffer variable examp_file^. To place this value in the second component, we call put:

```
append(examp_file);
examp_file^:= 9;
```

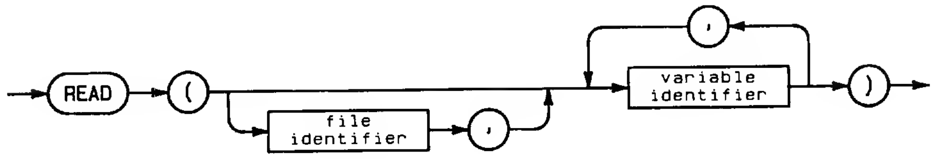


```
put(examp_file):
```



read

This procedure assigns the value of the current component of a file to its arguments.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	file must be open to read; default = `output`
variable identifier	type compatible with file type; see semantics	-

Examples

```
read(file_var,variable)
read(file,variable1,...,variablen)
read(variable)
read(variable1,...,variablen)
```

Semantics

The procedure `read`(f,v) assigns the value of the current component of f to the variable v, advances the current position, and causes any subsequent reference to the buffer variable f^ to actually load the buffer with the new current component.

Variable Compatability

If the file is a textfile, the variable can be a simple, string, or PAC variable. If the file is not a textfile, its components must be assignment compatible with the variable. Any number of variable identifiers can appear separated by commas.

The parameter v may be a component of a packed structure.

The following statement:

```
read(f,v)
```

is equivalent to

```
v := f^
get(f);
```

If f is a textfile, an implicit data conversion may precede the `read` operation (see below).

The call

```
read(f,v1,...,vn);
```

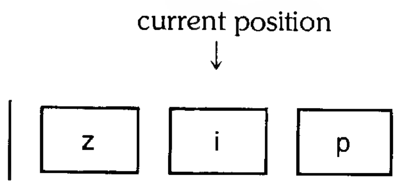
is equivalent to

```
read(f,v1);
read(f,v2);
  .
  .
  .
read(f,vn);
```

Illustration

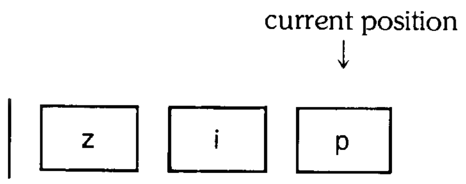
Suppose `examp_file` is a file of `char` opened in the read-only state. The current position is at the second component. To read the value of this component into `char_var`, we call `read`:

{initial condition}



state: read-only
examp_file^: i or undefined
eof (examp_file): false
char_var: old value, if any

read(examp_file,char_var)



state: read-only
examp_file^(deferred): p
eof(examp_file): false
char_var: i

Implicit Data Conversion

If f is a textfile, its components are type `char`. The parameter v, however, need not be type `char`. It may be any simple, `string`, or PAC type. The `read` procedure performs an implicit conversion from the ASCII form which appears in the textfile f to the actual form stored in the variable v.

If v is type `real`, `longreal`, `integer`, or an integer subrange, the `read` operation searches f for a sequence of characters which satisfies the syntax for these types. The search skips preceding blanks or end-of-line markers. If v is `longreal`, the result is independent of the letter preceding the scale factor.

An error occurs if the `read` operation finds no non-blank characters or a faulty sequence of characters, or if an integer value is outside the range of v. After `read`, a subsequent reference to the buffer variable f^ will actually load the buffer with the character immediately following the number read. Also note that `eof` will be `false` if a file has more blanks or line markers, even though it contains no more numeric values.

If v is a variable of type `string` or PAC, then `read`(f,v) will fill v with characters from f. When v is type PAC and `eoln`(f) becomes `true` before v is filled, the operation puts blanks in the rest of v. If v is type string and `eoln`(f) becomes `true` before v is filled to its maximum length, no blank padding occurs. `Strlen`(v) then returns the actual number of characters in v. You may wish to use this fact to determine the actual length of a line in a textfile.

If v is a variable of an enumerated type, `read`(f,v) searches f for a sequence of characters satisfying the syntax of a HP Pascal identifier. The search skips preceding blanks and line markers. Then the operation compares the identifier from f with the identifiers which are values of the type of v, ignoring upper and lower case distinctions. Finally, it assigns an appropriate value to v. An error occurs if the search finds no non-blank characters, if the string from f is not a valid HP Pascal identifier, or if the identifier doesn't match one of the identifiers of the type of v.

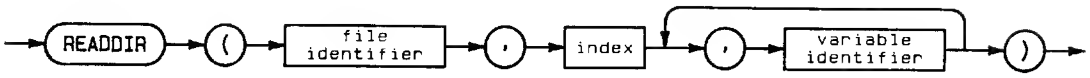
The following table shows the results of calls to `read` with various sequences of characters for different types of v.

Implicit Data Conversion

Sequence of characters in f following current position	Type of v	Result stored in v
(space)(space)1.850	real	1.850
space)(linemarker)(space)1.850	longreal	1.850
10000(space)10	integer	10000
8135(end-of-line)	integer	8135
54(end-of-line)36	integer	54
1.583E7	real	1.583x10(7)
1.583E+7	longreal	1.583x10(7)
(space)Pascal	string[5]	'Pasc'
(space)Pas(end-of-line)cal	string[9]	'Pas'
(space)Pas(end-of-line)cal	PAC {length 9}	'Pas'
(end-of-line)Pascal	PAC {length 5}	'Pasca'
(space)Monday(space)	ennumerated	Monday

readdir

This procedure reads a specified component from a direct-access file.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	file must be open to read; file must not be a textfile
index	integer expression	greater than 0; less than `lastpos`(file identifier)
variable identifier	variable that is type compatible with file type	see semantics

Examples

```
readdir(file_var,indx,variable)
readdir(file_var,indx,variable1,...,variablen)
```

Semantics

The procedure `readdir`(f,k,v) places the current position at component k and then reads the value of that component into v. Formally, this is equivalent to:

```
seek(f,k);
read(f,v);
```

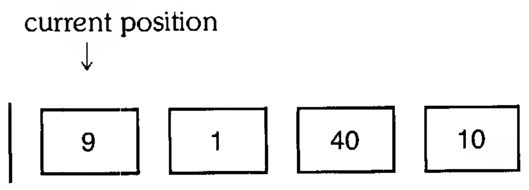
The call `get`(f) is not required between `seek` and `read` because of the definition of `read`.

You can use the procedure `readdir` only with files opened for direct access. Thus, a textfile cannot appear as a parameter for `readdir`.

Illustration

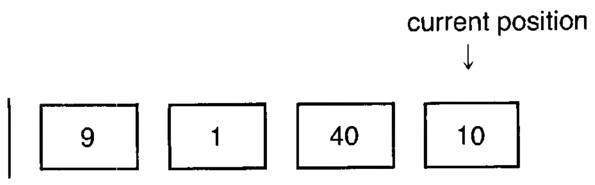
Suppose examp_file is a file of `integer` with four components opened in the read-write state. The current position is the first component. To read the third component into int_var, we call `readdir`. After `readdir` executes, the current position is the fourth component.

{initial condition}



state: read-write
examp_file^: undefined
eof(examp_file): false
int_var: old value

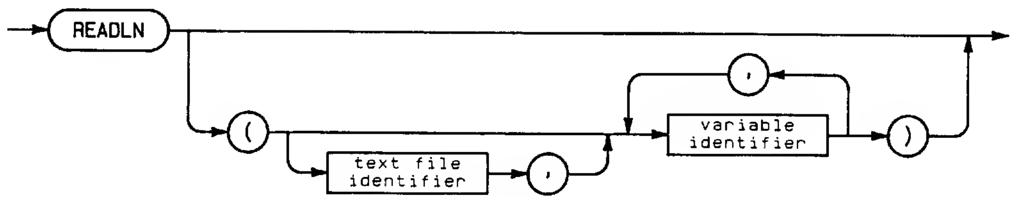
readdir(examp_file,3,int_var);



state: read-write
examp_file^(deferred): 10
eof(examp_file): false
int_var: 40

readln

This procedure reads a value from a textfile and then advances the current position to the beginning of the next line.



Item	Description/Default	Range Restrictions
textfile identifier	variable of type textfile; default = `input`	file must be open to read
variable identifier	variable must be a simple type, a string type or a PAC	-

Examples

```
readln(file)
readln(file,variable)
readln(file,variable1,...,variablen)
readln(variable)
readln(variable1,...,variablen)
readln
```

Semantics

The procedure `readln`(f,v) reads a value from the textfile f into the variable v and then advances the current position to the beginning of the next line, i.e. the first character after the next end-of-line marker. The operation performs implicit data conversion if v is not type `char` (see discussion of `read` above).

The call `readln(f,v1,...,vn)` is equivalent to

```
read(f,v1,...,vn);
readln(f);
```

If the parameter v is omitted, `readln` simply advances the current position to the beginning of the next line.

real

The type `real` represents a subset of the real numbers.

The type `real` is a standard simple type. For HP Pascal, the range of the subset is implementation dependent.

Permissible Operators

assignment:	`:=`
relational:	`<, <=, =, <>, >=, >`

`arithmetic - +, -, *, /`

Standard Functions

real argument:	`abs, arctan, cos, exp, ln, round, sin, sqr, sqrt, trunc`
real return:	`abs, arctan, cos, exp, ln, sin, sqr, sqrt`

Example Code

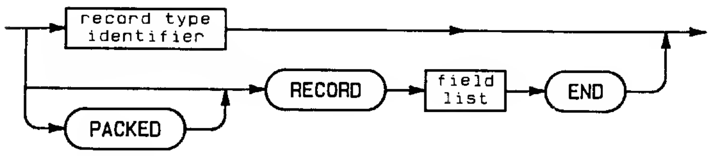
```
PROGRAM show_realnum(output);

VAR
  realnum: real;

BEGIN
  realnum := 6.023E+23;
  writeln(realnum);
END.
```

RECORD

A record is a collection of components which are not necessarily the same type. Each component is termed a field of the record and has its own identifier.

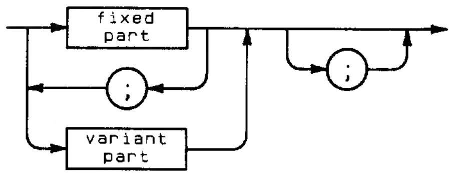
A record type is a structured type and consists of the reserved word RECORD, a field list, and the reserved word END.

The reserved word PACKED may precede the reserved word RECORD. It instructs the compiler to optimize storage of the record fields.



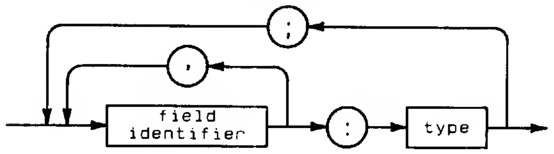
The field list has a fixed part and an optional variant part.

Field List:



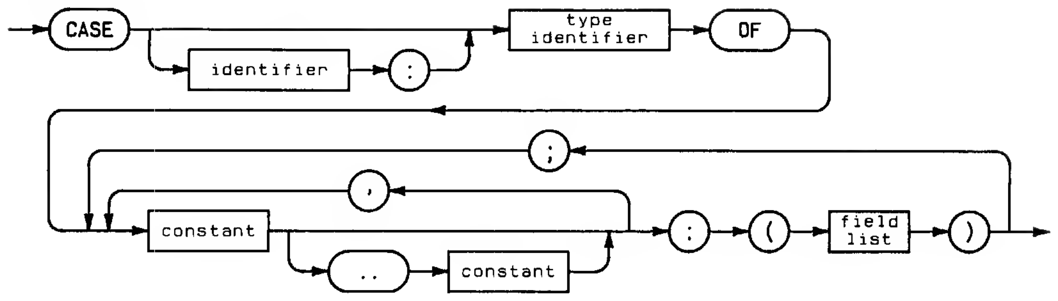
In the fixed part of the field list, a field definition consists of an identifier, a colon (:), and a type. Any simple, structured, or pointer type is legal. Several fields of the same type can be defined by listing identifiers separated by commas.

Fixed Part of a Field List:



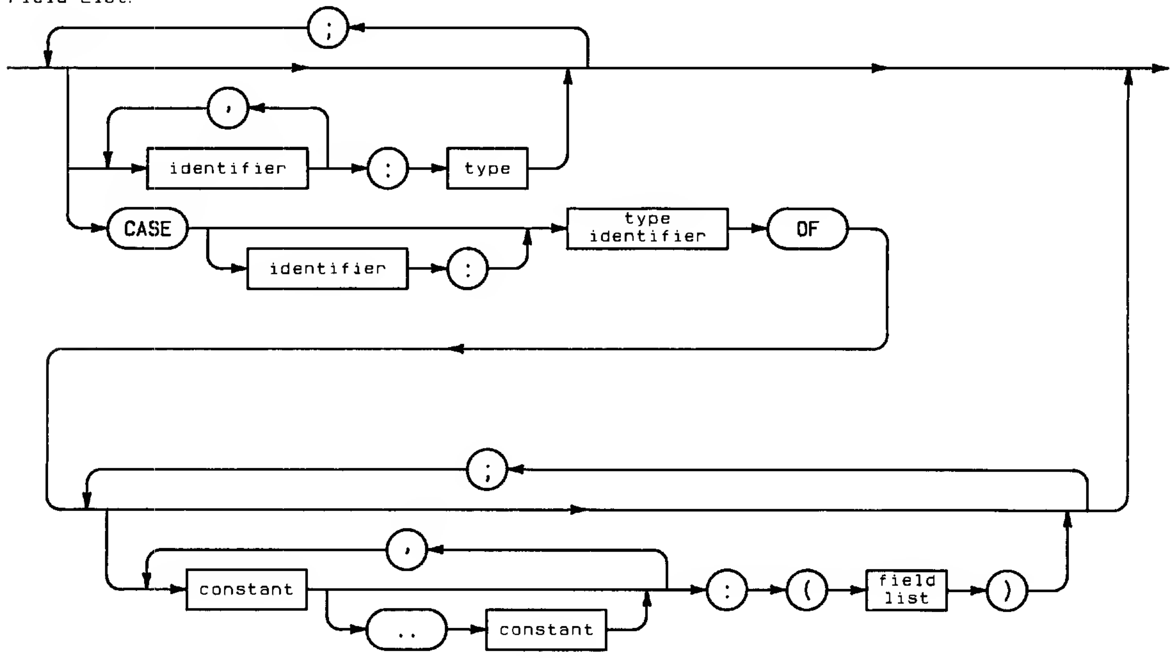
In the variant part, the reserved word CASE introduces an optional tag field identifier and a required ordinal type identifier. Then the reserved word OF precedes a list of case constants and alternative field lists. Fields of type file or of a type which contains files are not legal in the variant part of a record.

Variant Part of a Field List:



Case constants must be type compatible with the tag. Several case constants may be associated with a single field list. The various constants appear separated by commas. Subranges are also legal case constants. The empty field list may be used to indicate that a variant doesn't exist (see example). HP Pascal does **not** require that you specify all possible tag values.

Field List:



You may not use the OTHERWISE construction in the variant part of the field list. OTHERWISE is only legal in CASE statements.

Variant parts allow variables of the same record type to exhibit structures that differ in the number and type of their component parts. If a record has multiple variants, when a variant is assigned to the tag field, any fields associated with a previous variant cease to exist and the new variant's fields come into existence with undefined values. An error occurs if a reference is made to a field of a variant other than the current variant.

A field of a record is accessed by using the appropriate field selector.

Permissible Operators

assignment (entire record):	:=
field selection:	.

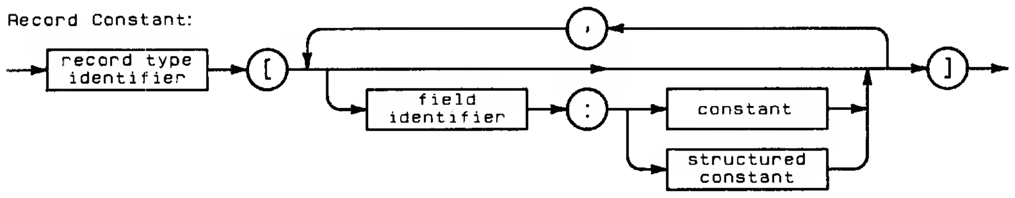
Example Code

```
TYPE
  word_type = (int, ch);
  word      = RECORD              {variant part only with tag}
                CASE word_tag: word_type OF
                  int: (number: integer);
                  ch : (chars : PACKED ARRAY [1..2] OF char);
              END;

  polys   = (circle, square, rectangle, triangle);
  polygon = RECORD        {fixed part and tagless variant part}
              poly_color: (red, yellow, blue);
              CASE polys OF
                circle:    (radius: integer);
                square:    (side: integer);
                rectangle: (length, width: integer);
                triangle:  (base, height: integer);
            END;

  name_string  = PACKED ARRAY [1..30] OF char;
  date_info    = PACKED RECORD              {fixed part only}
                   mo: (jan, feb, mar, apr, may, jun,
                        jul, aug, sep, oct, nov, dec);
                   da: 1..31;
                   yr: 1900..2001;

                 END;
  marital_status = (married, separated, divorced, single);
  person_info    = RECORD                {nested variant parts}
                     name: name_string;
                     born: date_info;
                     CASE status: marital_status OF
                       married..divorced:
                                  (when:   date_info;
                                   CASE has_kids: boolean OF
                                     true: (how_many: 1..50);
                                     false: (); {Empty variant}
                                  );
                       single: ();
                   END;
```

Record Constructor

A record constant is a declared constant defined with a record constructor which specifies values for the fields of a record type.

A record constructor consists of a previously declared record type identifier and a list in square brackets of fields and values. All fields of the record type must appear, but not necessarily in the order of their declaration. Values in the constructor must be assignment compatible with the fields.



For records with variants, the constructor must specify the tag field before any variant fields. Then only the variant fields associated with the value of the tag may appear. For free union variant records, i.e. tagless variants, the initial variant field selects the variant.

The values may be constant values or constructors. To use a constructor as a value, you must define the field in the record type with a type identifier. A record constant may not contain a file.

A record constructor is only legal in the CONST section of a declaration part. It cannot appear in other sections or in an executable statement.

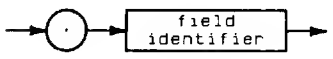
A record constant may be used to initialize a variable in the body of a block. You can also select individual fields of a record constant in the body of a block, but not when defining other constants.

Example Code

```
TYPE
  securtype = (light, medium, heavy);
  counter   = RECORD
                pages: integer;
                lines: integer;
                characters: integer;
              END;
  report    = RECORD
                revision: char;
                price:    real;
                info:     counter;
                CASE securtag: securtype OF
                  light:   ();
                  medium:  (mcode: integer);
                  heavy:   (hcode:    integer;
                            password: string[10]);
              END;

 CONST
  no_count   = counter [pages: 0, characters: 0, lines: 0];
  big_report = report [revision: 'B',
                       price:     19.00,
                       info:      counter [pages:      19,
                                           lines:      25,
                                           characters: 900],
                       securtag:  heavy,
                       hcode:     999,
                       password:  'unity'];


no_report  =  report [ revision : ' ';
                       price    : 0.00;
                       info     : no_count;
                       securtag : light];
```

Record Selector

A record selector accesses a field of a record. The record selector follows a record designator and consists of a period and the name of a field.



A record designator is the name of a record, the selected component of a structure which is a record, or a function call which returns a record.

The WITH statement "opens the scope" of a record, making it unnecessary to specify a record selector.

Example Code

```
PROGRAM show_recordselector;
TYPE
  r_type = RECORD
             f1: integer;
             f2: char;
           END;
 VAR
  a,b       : integer;
  ch        : char;
  r         : r_type;
  rec_array : ARRAY [1..10] OF r_type;
BEGIN
   .
  a:= r.f1 + b;       {Assigns current value of integer field  }
   .                  {of r plus b to a.                       }
   .
  rec_array[a].f2:= ch; {Assigns current value of ch to char   }
   .                    {field of a'th component of rec_array.}
END.
```

Recursion

A recursive procedure or function is a procedure or function that calls itself. It is also legal for procedure A to call procedure B which in turn calls procedure A. This is indirect recursion and is often an instance when the FORWARD directive is useful.

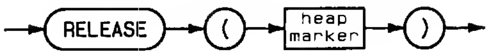
When a routine is called recursively, new local variables are created dynamically (on the stack).

Example Code

```
FUNCTION factorial (n: integer): integer;
{Calculates factorial recursively}
  BEGIN
    IF n = 0 THEN
      factorial := 1
     ELSE
      factorial := n * factorial(n-1);
  END;
```

release

This procedure returns the heap to its state when it was marked by the `mark` procedure.



Item	Description/Default	Range Restrictions
heap marker	a pointer variable	pointer should have previously appeared as a parameter in a call to `mark`, and should not have been passed to `release` see semantics

Example

```
release(ptr)
```

Semantics

The procedure `release`(p) returns the heap to its state when `mark` was called with p as a parameter. This has the effect of deallocating any heap variables allocated since the program called `mark`(p). The system can then reallocate the released space. The system automatically closes any files in the released area.

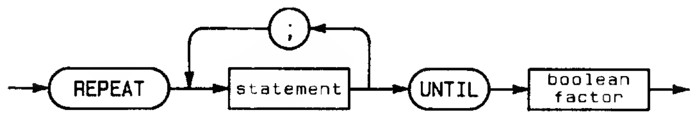
An error occurs if p is not passed as a parameter to `mark`, or if it was previously passed to `release` explicitly or implicitly (see example below). After `release`, p is undefined.

Example Code

```
PROGRAM show_markrelease;
VAR
 w,x,y: ^integer;
BEGIN
  .
  mark(w);
  .
  release(w);  {Returns heap to state marked by w.      }
  .
  mark(x);
  .
  mark(y);
  .
  release(x);  {Returns heap to state marked by x. The }
  .            {pointer y no longer marks a heap state.}
END.           {Release(y) is now an error.            }
```

REPEAT

A REPEAT statement executes a statement or group of statements repeatedly until a given condition is true.



A REPEAT statement consists of the reserved word REPEAT, one or more statements, the reserved word UNTIL, and a boolean factor (the condition).

The statements between REPEAT and UNTIL need not be bracketed with BEGIN..END.

When the system executes a REPEAT statement, it first executes the statement sequence and then evaluates the condition. If it is false, it executes the statement sequence and evaluates the condition again. If it is true, control passes to the statement after the REPEAT statement.

The statement

```
REPEAT
  statement;
UNTIL condition
```

is equivalent to the following:

```
1: statement;
   IF NOT condition THEN GOTO 1;
```

Usually the statement sequence will modify data at some point so that the condition becomes false. Otherwise, the REPEAT statement will loop forever. Of course, it is possible to branch unconditionally out of a REPEAT statement using a GOTO statement.

The compiler can be directed to perform partial evaluation of boolean operators used in a REPEAT...UNTIL statement. For example:

```
REPEAT ... UNTIL done OR finished
```

By specifying the $PARTIAL_EVAL ON$ compiler directive, if "done" is `true`, the remaining operators will not be evaluated since execution of the statement depends on the logical OR of both operators. (Both operators would have to be `false` for the logical OR of the operators to be `false`.)

Example Code

```
sum := 0;
count := 0;
  REPEAT
    writeln('Enter trial value, or "-1" to quit');
    read (value);
    sum := sum + value;
    count := count + 1;
    average := sum / count;
    writeln ('value =', value, '    average =', average)
  UNTIL (count >= 10) OR (value = -1);
  .
  .
REPEAT
  writeln (real_array [index]);
  index := index + 1;
UNTIL index > limit;
```

Reserved Words

These are the reserved words recognized by HP Pascal.

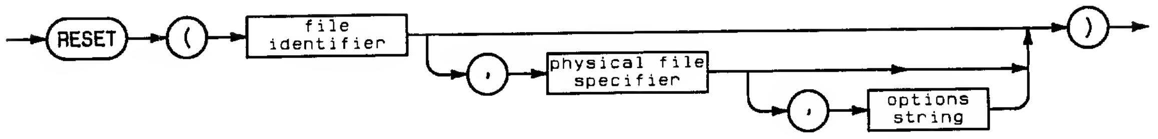
```
AND         ARRAY

BEGIN

CASE        CONST

DIV         DO          DOWNTO

ELSE        END         EXPORT

FILE        FOR         FUNCTION

GOTO

IF          IMPLEMENT   IMPORT      IN

LABEL

MOD         MODULE

NIL         NOT

OF          OR          OTHERWISE

PACKED      PROCEDURE   PROGRAM

RECORD      REPEAT

SET

THEN        TO          TYPE

UNTIL

VAR

WHILE       WITH
```

Reserved words can **not** be used as identifiers.

The letter-case of reserved words is unimportant. They may be typed in either upper or lower case.

reset

This procedure opens a file in the read-only state and places the current position at the first component.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	-
physical file specifier	name to be associated with f; must be a string expression or PAC variable	-
options string	a string expression or PAC variable	implementation dependent

Examples

```
reset(file_var)
reset(file_var,file_name)
reset(file_var,file_name,opt_str)
```

Semantics

The procedure `reset`(f) opens the file f in the read-only state and places the current position at the first component. The contents of f, if any, are undisturbed. The file f may then be read sequentially.

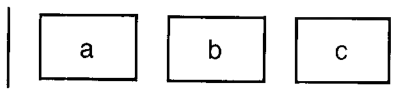
If f is not empty, `eof`(f) is `false` and a subsequent reference to the buffer variable f^ will actually load the buffer with the first component. The components of f may now be read in sequence. If f is empty, however, `eof`(f) is `true` and f^ is undefined. A subsequent call to `read` produces an error.

If f is already open at the time `reset` is called, the system automatically closes and then reopens it. If the parameter s is specified, the system closes any physical file previously associated with f.

Illustration

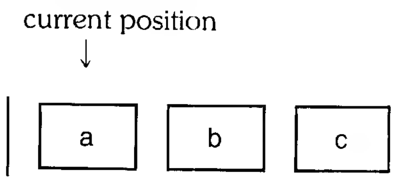
Suppose examp_file is a closed file of `char` with three components. To read sequentially from examp_file, we call `reset`:

{initial condition}



state: closed

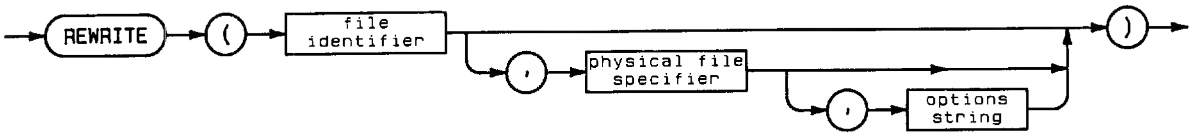
reset(examp_file);



state: read-only
examp_file^(deferred): a
eof(examp_file): false

rewrite

This procedure opens a file in the write-only state and places the current position at the beginning of the file.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	-
physical file specifier	name to be associated with f; must be a string expression or PAC variable	-
options string	a string expression or PAC variable	implementation dependent

Examples

```
rewrite(file)
rewrite(file,file_name)
rewrite(file,file_name,opt_str)
```

Semantics

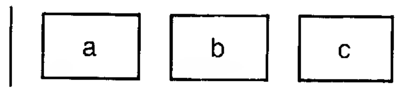
The procedure `rewrite`(f) opens the file f in the write-only state and places the current position at the beginning of the file. The system discards any previously existing components of f. The function `eof`(f) returns `true` and the buffer variable f^ is undefined. You may now write on f sequentially.

If f is already open at the time `rewrite` is called, the system closes it automatically and then reopens it. If s is specified, the system closes any physical file previously associated with f.

Illustration

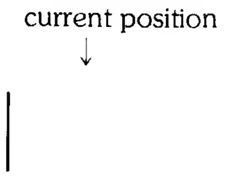
Suppose examp_file is a closed file of `char` with three components. To discard these components and write sequentially to examp_file, we call `rewrite`:

{initial condition}



state: closed

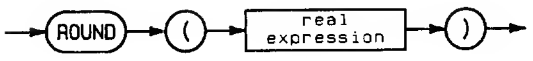
rewrite(examp_file);



state: write-only
examp_file^ : undefined
eof(examp_file): true

round

This function returns the argument rounded to the nearest integer.



Examples

Input	Result
`round(bad_real)`	
`round(3.1)`	3
`round(-6.4)`	-6
`round(-4.6)`	-5
`round(1.5)`	2

Semantics

The function `round`(x) returns the integer value of x rounded to the nearest integer. If x is positive or zero, then `round`(x) is equivalent to `trunc`(x + 0.5); otherwise, `round`(x) is equivalent to `trunc`(x − 0.5). An integer overflow occurs if the result is not in the range `minint..maxint`.

Scope

The scope of an identifier is its domain of accessibility, i.e. the region of a program in which it may be used.

In general, a user-defined identifier may appear anywhere in a block after its definition. Furthermore, the identifier may appear in a block nested within the block in which it is defined.

If an identifier is redefined in a nested block, however, this new definition takes precedence. The object defined at the outer level will no longer be accessible from the inner level (see example below).

Once defined at a particular level, an identifier may not be redefined at the same level (except for field names).

Labels are not identifiers and their scope is restricted. They cannot mark statements in blocks nested within the block where they are declared.

Identifiers defined at the main program level are "global". Identifiers defined in a function or procedure block are "local" to the function or procedure.

The definition of an identifier must precede its use, with the exception of pointer type identifiers, program parameters, and forward declared procedures or functions.

For a module, identifiers declared in the EXPORT section are valid for the entire module, identifiers declared after the IMPLEMENT keyword are valid only within the module.

Example Code

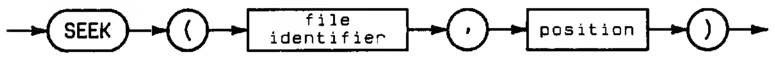
```
PROGRAM show_scope (output);
CONST
  asterisk = '*';
 VAR
  x: char;
PROCEDURE writeit;
  CONST
    x = 'LOCAL AND GLOBAL IDENTIFIERS DO NOT CONFLICT';
  BEGIN
    write (x)
  END;
BEGIN {show_scope}
  x:= asterisk;
  write (x);
  writeit;
  write (x)
END.  {show_scope}
```

Results:

```
*LOCAL AND GLOBAL IDENTIFIERS DO NOT CONFLICT*
```

seek

This procedure places the current position of a file at the specified component.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	must be direct access; must be open for read-write
index	integer expression	greater than 0

Example

```
seek(file_var,indx)
```

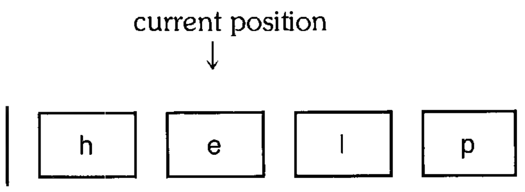
Semantics

The procedure `seek`(f,k) places the current position of f at component k. If k is greater than the index of the highest-indexed component ever written to f, the function `eof`(f) returns `true`, otherwise `false`. The buffer variable f^ is undefined following the call to `seek`. An error occurs if f is not open in the read-write state.

Illustration

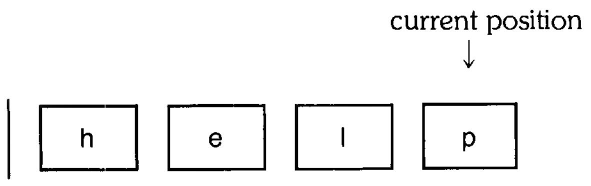
Suppose examp_file is a file of `char` with four components opened for direct access. The current position is the second component. To change it to the fourth component, we call `seek`.

{initial condition}



state: read-write
examp-file^(deferred): e
eof(examp_file): false

seek(examp_file,4);



state: read-write
examp_file^: undefined
eof(examp_file): false

Separators

A separator is a blank, an end-of-line marker, a comment, or a compiler option.

At least one separator must appear between any pair of consecutive identifiers, numbers, or reserved words. When one or both elements are special symbols, however, the separator is optional.

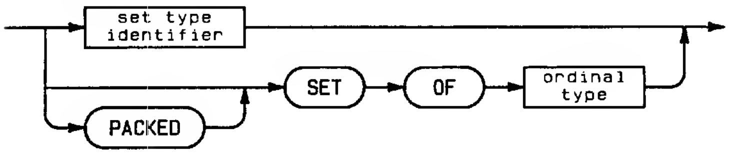
Example Code

```
IF eof THEN GOTO 99           {Required separators.}
x := x + 1                    {Optional separators.}
x:=x+1                        {No separators.      }
```

SET

A set is the powerset, i.e. the set of all subsets, of a base type. A set type consists of the reserved words SET OF and an ordinal base type.

Set Type:



A set type is a user-defined structured type. The base type may be any ordinal type. The maximum number of elements is implementation defined but must be at least 256 elements. It is legal to declared a packed set, but whether this affects storage is implementation dependent.

Permissible Operators

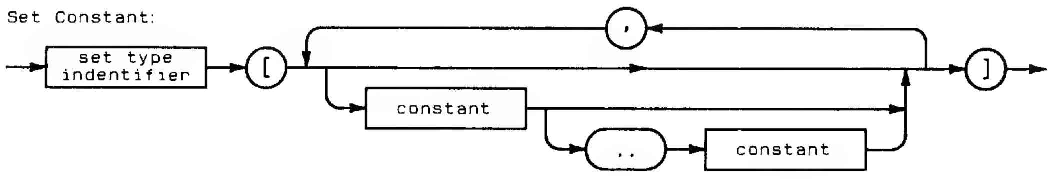
assignment:	:=
union:	+
intersection:	*
difference:	-
subset:	<=
superset:	>=
equality:	=, <>
inclusion:	IN

Example Code

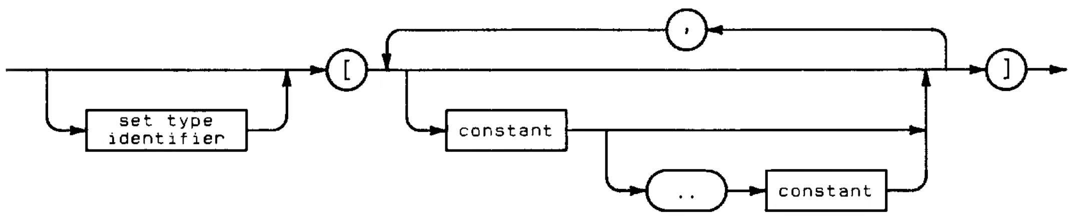
```
TYPE
  charset   = SET OF char;
  fruit     = (apple, banana, cherry, peach, pear, pineapple);
  somefruit = SET OF apple..cherry;
  poets     = SET OF (Blake, Frost, Brecht);
  some_set  = SET OF 1..200;
```

Restricted Set Constructor

A set constant is a declared constant defined with a restricted set constructor which specifies set values.



A restricted set constructor consists of an optional previously declared set type identifier and a list of constant values in square brackets. Subranges may appear in this list.



A value must be an ordinal constant value or an ordinal subrange. A constant expression is legal as a value. The symbols (. and .) may replace the left and right square brackets, respectively.

Restricted set constructors may appear in a CONST section of a declaration part or in executable statements. Unrestricted set constructors permit variables to appear as values within the brackets.

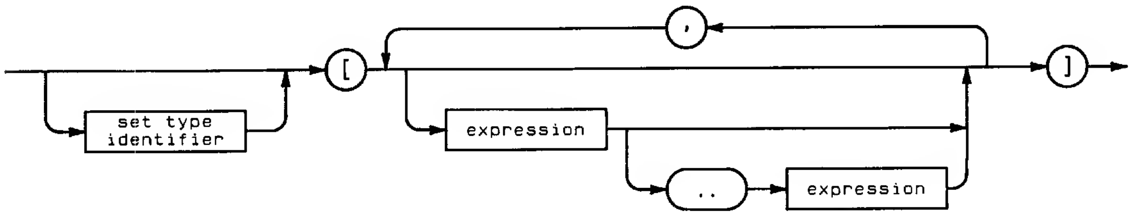
You can use a set constant to initialize a set variable in the body of a block.

Example Code

```
TYPE
  digits  = SET OF 0..9;
  charset = SET OF char;
CONST
  all_digits = digits [0..9];              {Subrange.}
  odd_digits = digits [1, 1+2, 5, 7, 9];
  letters    = charset ['a'..'z', 'A'..'Z'];
  no_chars   = charset [];
  no_iden    = [2, 4, 6, 8]         {No set identifier.}
```

Set Constructor

A set constructor designates one or more values as members of a set whose type may or may not have been previously declared. A set constructor consists of an optional set type identifier and one or more ordinal expressions in square brackets. Two expressions may serve as the lower and upper bound of a subrange.



If the set type identifier is specified, the values in the brackets must be type compatible with the base type of the set. If no set type identifier appears, the values must be type compatible with each other. The symbols (. and .) may replace the left and right square brackets, respectively.

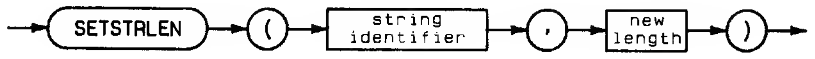
Set constructors may appear as operands in expressions in executable statements. Set constructors with constant values are legal in the definition of constants.

Example Code

```
PROGRAM show_setconstructor;
TYPE
  int_set = SET OF 1..100;
  cap_set = SET OF 'A'..'Z';
VAR
  a,b: 0..255;
  s1: SET OF integer;
  s2: SET OF char;
BEGIN
  .
  .
  s1:= int_set[(a MOD 100) + (b MOD 100)]
  s2:= cap_set['B'..'T', 'X', 'Z'];
END.
```

setstrlen

This procedure sets the current length of s to the specified length.



Item	Description/Default	Range Restrictions
string identifier	variable of type string	-
new length	integer expression	0 thru the maximum length of the string

Example

```
setstrlen(str_var,int_exp)
```

Semantics

The procedure `setstrlen`(s,e) sets the current length of s to e without modifying the contents of s.

If the new length of s is greater than the previous length of s, the extra components will be undefined. No blank filling occurs. If the new length of s is less than the previous length of s, previously defined components beyond the new length will no longer be accessible.

Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

```
VAR
  alpha: string[80];
BEGIN
   .
  alpha:= 'abcdef';       {strlen(alpha) = 6}
   .
  setstrlen(alpha,2*strlen(alpha)); {Doubles current length }
   .                                {of alpha. Alpha[7]     }
   .                                {through alpha[12] not  }
   .                                {defined.               }
   .
  setstrlen(alpha,2)                {Alpha[3] through       }
   .                                {alpha[80] unavailable. }
END.
```

Side Effects

A side effect is the modification, by a procedure or function, of a variable not appearing in the parameter list.

Global variables are declared at the beginning of a program before any procedure declarations. Global variables are valid during the execution of the program.

Local variables are variables declared within a procedure or function (or in the headings as parameters) and are only valid during the execution of the procedure of function.

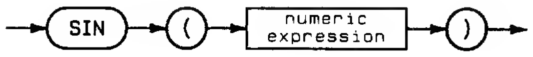
If you declare a local variable using the same identifier as a global variable, the local variable can be modified without affecting the global variable. A side effect is likely to occur if you forget to declare the variable within the procedure or the procedure heading. Without the local declaration, the compiler assumes that the global variable is to be used.

Example Code

```
PROGRAM show_effects(output);

VAR i,j : integer;             {Global variables}

PROCEDURE oops(i : integer);  {i is local to the procedure}

  BEGIN
    IF i > 0 THEN j := j - 1;  {j is a global variable}
  END;

BEGIN
  i := 2;
  j := 3;
  oops(i);
  IF i = j THEN writeln('There was a side effect');
END.
```

sin

This function returns the sine of the angle represented by its argument.



Examples

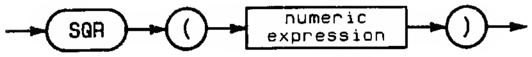
Input	Result
`sin(rad)`	
`sin(0.024)`	`2.399770E-02`

Semantics

The function `sin`(x) computes the sine of x, where x is interpreted to be in radians. X can be any numeric value.

sqr

This function computes the square of its argument.



Examples

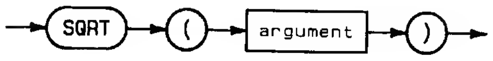
Input	Result
`sqr(3)`	`9`
`sqr(1.198E3)`	`1.435204E+06.`
`sqr(maxint)`	`{error}`

Semantics

The function `sqr`(x) computes the value of x squared. If x is an integer value, the result is also an integer. If the value to be returned is greater than the maximum value for a particular type, a run-time error occurs.

sqrt

This function computes the square root of its argument.



Item	Description/Default	Range Restrictions
argument	numeric expression	greater than or equal to 0

Examples

```
Input                   Result
sqrt(64)                8.000000E+00
sqrt(13.5E12)           3.674235E+06
sqrt(0)                 0.000000E+00
sqrt(-5)                {error}
```

Semantics

The function sqrt(x) computes the square root of x. If x is less than 0, a run-time error occurs.

Standard Procedures and Functions

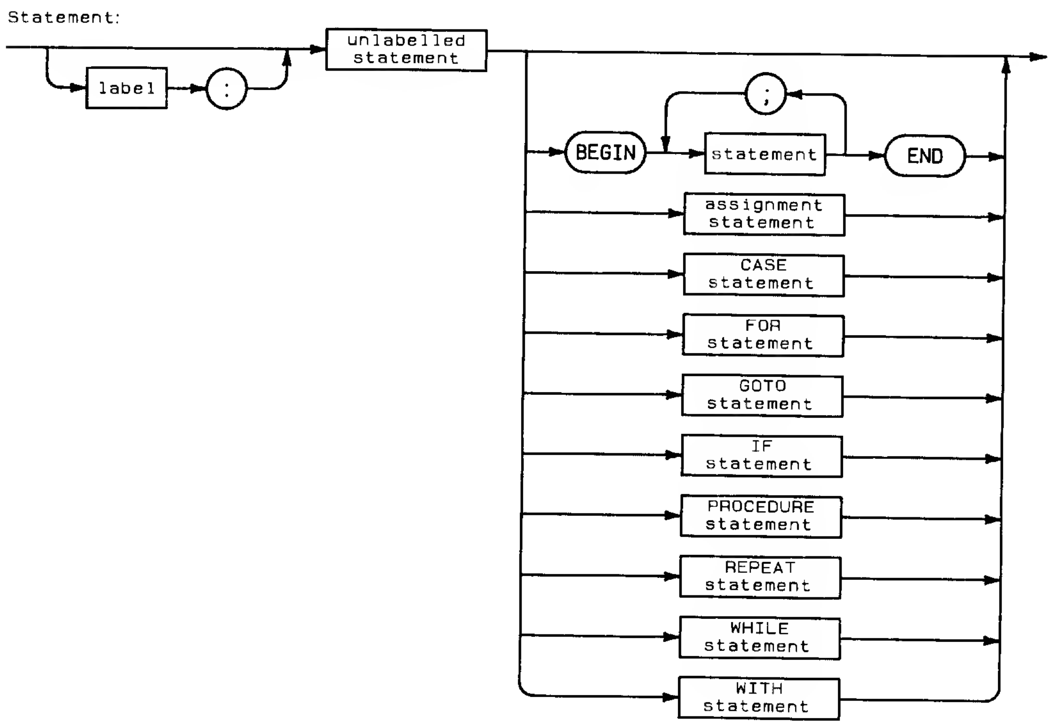
The standard procedures and functions recognized by HP Pascal are listed in the following tables. These identifiers may be redefined within a program since they appear "global" to a program.

Standard Procedures and Functions for HP Pascal

Procedures	Functions
`append`	`abs`
`close`	`arctan`
`dispose`	`binary`
`get`	`chr`
`halt`	`cos`
`mark`	`eof`
`new`	`eoln`
`open`	`exp`
`overprint`	`hex`
`pack`	`lastpos`
`page`	`linepos`
`prompt`	`ln`
`put`	`maxpos`
`read`	`octal`
`readdir`	`odd`
`readln`	`ord`
`release`	`position`
`reset`	`pred`
`rewrite`	`round`
`seek`	`sin`
`setstrlen`	`sqr`
`strappend`	`sqrt`
`strdelete`	`str`
`strinsert`	`strlen`
`strmove`	`strmax`
`strread`	`strltrim`
`strwrite`	`strpos`
`unpack`	`strrpt`
`write`	`strrtrim`
`writedir`	`succ`
`writeln`	`trunc`

Statements

A statement is a sequence of special symbols, reserved words, and expressions which either performs a specific set of actions on data or controls program flow.



HP Pascal statement types and purposes include:

Statement Type	Purpose
compound	group statements
empty	do nothing
assignment	assign a value to a variable
procedure	activate a procedure
GOTO	transfer control unconditionally
IF, CASE	conditional selection
WHILE, REPEAT, FOR	iterate a group of statements
WITH	manipulate record fields

Empty, assignment, procedure, and GOTO statements are "simple" statements. IF, CASE, WHILE, REPEAT, FOR, and WITH statements are "structured" statements because they themselves may contain other statements.

A GOTO statement requires a label to mark the location of the statement where execution is to continue. The label consists of an unsigned integer and a colon ":" preceeding the "target" statement. When a label is used, a LABEL declaration must appear in the declaration section of the block containing the GOTO statement and its destination statement.

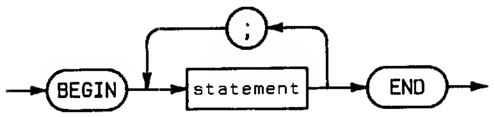
The following pages describe compound, and empty statements.

Compound Statements

A compound statement is a sequence of statements bracketed by the reserved words BEGIN and END. A semi-colon (;) delimits one statement from the next. The system executes the sequence of statements in order.

Certain statements may alter the flow of execution in order to achieve effects such as selection, iteration, or invocation of another procedure or function.

After the last statement in the body of a routine has executed, control is returned to the point in the program from which the routine was called. The program terminates after the last statement is executed.



A compound statement has two primary uses: (1) it defines the statement part of a block; (2) it replaces a single statement within a structured statement. A compound statement may also serve to logically group a series of statements.

Compound statements are allowed but unnecessary in the following cases.

1. The statements between REPEAT and UNTIL
2. The statements between OTHERWISE and the end of the CASE statement.

Example Code

```
PROCEDURE check_min;
  BEGIN                                                   {This        }
    IF min > max THEN                                     {compound    }
      BEGIN                              {Compound      } {statement   }
        writeln('Min is wrong.');        {statement is} {is          }
        min := 0;                        {part of IF    } {the         }
      END;                               {statement.    } {procedure's}
  END;                                                    {body.       }
. . .

BEGIN                    {Nested compound statements        }
  BEGIN                  {for logically grouping statements.}
    start_part_1;
    finish_part_1;
   END;

  BEGIN
     start_part_2;
    finish_part_2;
   END;
END;
```

Empty Statements

An empty statement performs no action and is denoted by no symbol. It is often useful for indicating that nothing should occur or for inserting extra semi-colons in code.

These two statements, for example, explicitly specify no action when i is 2,3,4,6,7,8,9, or 10:

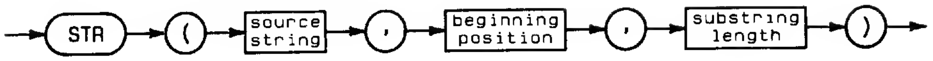
```
CASE i OF                              IF i IN [2..4, 6..10] THEN
  0     : start;                         {do nothing}
  1     : continue;                      ELSE continue;
  2..4  : ;
  5     : report_error;
  6..10: ;
   11    : stop;
  OTHERWISE fatal_error;
 END;
```

In this compound statement, there is an empty statement before END:

```
BEGIN
   I:= J + 1;
  K:= I + J;
 END
```

str

This function returns a portion of a string.



Item	Description/Default	Range Restrictions
source string	expression of type string	-
beginning position	integer expression	1 thru the current length of the string + 1
substring length	integer expression	0 thru 1 + the maximum length of the string − the beginning position

Example

```
str(str_exp,beg_pos,sub_len)
```

Semantics

The function `str`(s,b,e) returns the portion of s which starts at s[b] and is of length e. The result is type `string` and may be used as a string expression. An error occurs if strlen(s) is less than the sum of b and e minus 1, or b.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

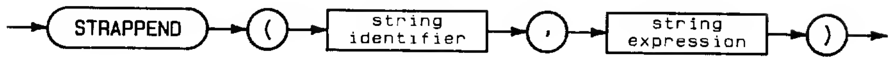
Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

```
VAR
   i: integer;
   wish_list: string[132];
  granted: string[5];
 BEGIN
   .
  i:= 13;
   wish_list:= 'wish1 wish2 wish3 wish4 wish5';
  granted:= str(wish_list,i,5);       {Selects the 3rd wish.}
   .                                  {Granted is 'wish3'.  }
 END.
```

strappend

This procedure appends one string to the end of another.



Item	Description/Default	Range Restrictions
string identifier	variable of type string	-
string expression	expression of type string	length must be less than the difference between the maximum and actual length of the string variable

Example

```
strappend(str_var,str_exp)
```

Semantics

The procedure `strappend`(s1,s2) appends string s2 to s1. The call passes s1 as an actual variable parameter to the procedure. The `strlen` of s2 must be less than or equal to `strmax`(s1) − `strlen`(s1). That is, it cannot exceed the number of characters left to fill in s1. The current length of s1 is updated to strlen(s1) + strlen(s2).

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

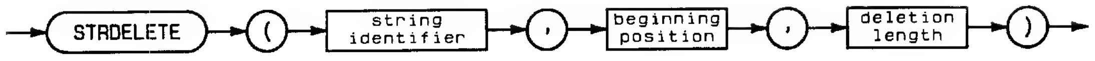
Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

```
VAR
  message: string[132]
BEGIN
   .
  message:= 'Now hear ';
  strappend(message,'this!');
   .
END.
```

strdelete

This procedure deletes characters from a string.



Item	Description/Default	Range Restrictions
string identifier	variable of type string	-
beginning position	integer expression	1 thru the current length of the string
deletion length	integer expression	0 thru 1 + the maximum length of the string − the beginning position

Example

```
strdelete(str_var,begin_pos,del_len)
```

Semantics

The procedure `strdelete`(s,p,n) deletes n characters from s starting at component s[p], and the current length of s is updated to the length s − n.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

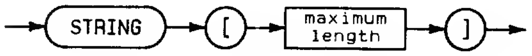
```
PROGRAM show_strdelete;
VAR
   long, short: string[80];
BEGIN
  long:= 'tiny pickle';
  strdelete(long,4,5);
  short:= long;           {short is 'tinkle'.}
END.
```

Strings

In HP Pascal, a string is a packed array of `char` whose maximum length is set at compile time and whose actual length may vary dynamically at run time.

A `string` type consists of the standard identifier `string` and an integer constant expression in square brackets which specifies the maximum length.

String Type:



Item	Description/Default	Range Restrictions
maximum length	integer expression	1 thru an implementation dependent number

The limit for the maximum length is implementation defined. The symbols (. and .) may replace the left and right square brackets, respectively.

A `String` type is a standard structured type.

Characters enclosed in single quotes are string literals. The compiler interprets a string literal as type PAC, `string`, or `char`, depending on context.

Integer constant expressions are constant expressions which return an integer value, an unsigned integer being the simple case (see Constant Definition above).

When a formal reference parameter is type `string`, you may choose not to specify the maximum length (see example below). This allows actual string parameters to have various maximum lengths.

A single component of a string can be accessed by using an integer expression in square brackets as a selector. The numbering of the characters in the string begins at one (1). In other words, to select the first character of a string named s, type: `s[1]`. The standard function `str` selects a substring of a string.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Note

Variables of string type, as other Pascal variables, are **not** initialized. The current string length contains meaningless information until you initialize the string.

Permissible Operators

assignment:	`:=`
concatenation:	`+`
relational:	`=, <>, <=, >=, >, <`

Standard Functions

string argument:	`str, strlen, strltrim, strmax, strpos, strrpt, strrtrim`
string return:	`str, strltrim, strrpt, strrtrim`

Standard Procedures

string parameter:	`setstrlen, strappend, strdelete, strinsert, strmove, strread, strwrite`

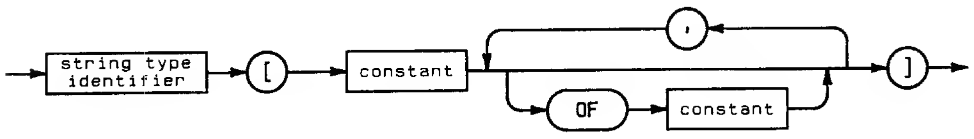
Example Code

```
CONST
  maxlength = 100;

TYPE
  name   = string[30];
  remark = string[maxlength * 2];

PROCEDURE proc1 (VAR s: string); EXTERNAL; {Maximum length }
                                           {not required.  }
```

String Constructor

A string constant is a declared constant defined with a string constructor which specifies values for a `string` type.

A string constructor consists of a previously defined string type identifier and a list of values in square brackets.



Within the square brackets, the reserved word OF indicates that a value occurs repeatedly. For example 3 OF 'a' assigns the character "a" to three successive string components. The symbols (. and .) may replace the left and right brackets, respectively. String literals of more than one character may appear as values.

The length of the string constant may not exceed the maximum length of the `string` type used in its definition.

String constructors are only legal in a CONST section of a declaration part. They cannot appear in other sections or in executable statements.

A string constant may be used to initialize a variable in the statement part of a block. You may also access individual components of a string constant in the body of the block, but not in the definition of other declared constants.

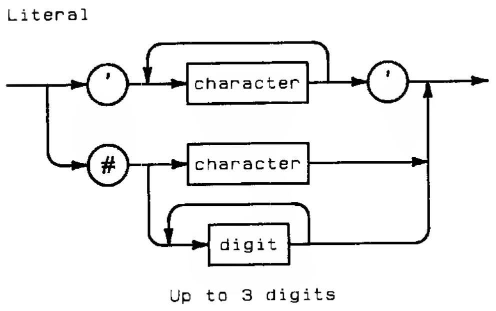
Example Code

```
TYPE
   s = string[80];

CONST
  blank = ' ';
  greeting = s['Hello!'];
  farewell = s['G',2 OF 'o','d','bye'];
  blank_string = s[10 OF blank];
```

String Literals

A string literal consists of any combination of the following.

- A sequence of ASCII printable characters enclosed in single quote marks.
- A sharp symbol (#) followed by a single character.
- A sharp symbol (#) followed by up to three digits which represent the ASCII value of a character.



The printable characters appearing between the single quotes are those ASCII characters assigned graphics and encoded by ordinal values 32 through 126.

A letter or symbol after a sharp symbol is equivalent to a control character. For example, #G or #g encodes CTRL-G, the bell character. The compiler interprets the letter or symbol according to the expression chr(ord(letter)MOD 32). Thus, the ordinal value of G is 71; modulus 32 of 71 is 7; and the ASCII value of 7 is the bell.

A number after a sharp symbol may contain up to three digits but must be in the range 0..255. It directly encodes any ASCII character, printing or non-printing. For example, the string literal #80#65#83#67#65#76 is equivalent to the string literal "PASCAL".

A string literal is type char, PAC or string, depending on the context.

If a single quote is a character in a string literal, it must appear twice.

A string literal may not be longer than a single line of source code, nor may it contain separators, except for spaces (blanks) within the quotes.

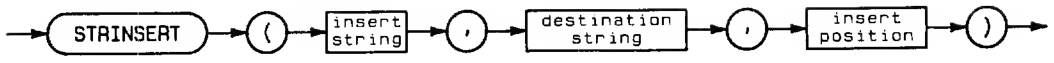
Two consecutive quote marks ('') specify the null or empty string literal. Assigning this value to a string variable sets the length of the variable to zero. Assigning it to a PAC variable blank-fills the variable.

Examples

```
'Please don''t!'                         {Single quote character.}
'A'
''                                       {Null string.            }
#F
#243#H
#27'that was an ESC char, and this is also'#[
'this string has five bells'#G#g#g#7#7' in it'
```

strinsert

This procedure inserts a string into another string.



Item	Description/Default	Range Restrictions
insert string	expression of type string	length less than maximum length of destination − insert position
destination string	variable of type string	-
insert position	integer expression	1 thru current length of destination string

Example

```
strinsert(insert,dest,pos)
```

Semantics

The procedure `strinsert`(s1,s2,n) inserts string s1 into s2 starting at s2[n]. Initially, s2 must be at least n-1 characters in length or an error will occur. The resulting string may not exceed `strmax`(s2). The current length of s2 is updated to strlen(s1) + strlen(s2).

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

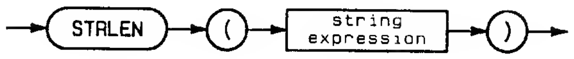
Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

```
VAR
  remark: string[80];
BEGIN
  .
  remark:= 'There is missing!';
  strinsert(' something',remark,9);
  .
END.
```

strlen

This function returns the current length of a string.



Example

```
strlen(str_exp)
```

Semantics

The function `strlen`(s) returns the current length of the string expression s.

If s is not initialized, `strlen`(s) is undefined.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Note

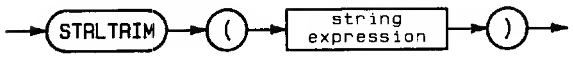
The `strlen` function can only be used with strings, not PAC's.

Example Code

```
VAR
   ars, vita: string[132];
  b: boolean;
BEGIN
    .
  IF strlen(ars) > strlen(vita) THEN
    b:= true
  ELSE
    halt;
    .
END.
```

strltrim

This function returns a string trimmed of all leading blanks.



Example

```
strltrim(str_exp)
```

Semantics

The function `strltrim`(s) returns a string consisting of s trimmed of all leading blanks. The function `strrtrim` trims trailing blanks.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

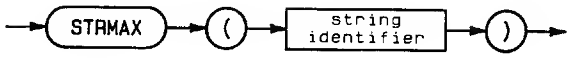
Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

```
VAR
   s: string[80];
BEGIN
   .
  s:= '            abc';
  s:=strltrim(s);          {s is now 'abc'}
   .                       {strlen(s) = 3 }
END.
```

strmax

This function returns the maximum allowable length of a string.



Item	Description/Default	Range Restrictions
string identifier	variable of type string	-

Example

```
strmax(str_var)
```

Semantics

The function `strmax`(s) returns the maximum length of s.

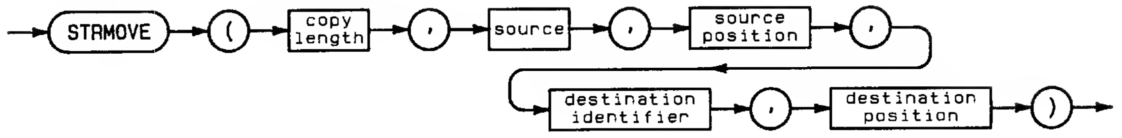
Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

```
VAR
  s: string[132];
BEGIN
   .
  IF strlen(s) = strmax(s) THEN
    BEGIN
      s:= strltrim(s);
      s:= strrtrim(s);
    END;
   .
END.
```

strmove

This procedure copies characters from one string or PAC to another.



Item	Description/Default	Range Restrictions
copy length	expression of type integer	see semantics
source	expression of type string or variable of type PAC	-
source position	integer expression	1 thru current length of source string
destination identifier	variable of type string or PAC	-
destination position	integer expression	1 thru current length of destination string − 1

Example

```
strmove(copy_len,source,source_pos,dest_id,dest_pos)
```

Semantics

The procedure `strmove`(n,s1,p1,s2,p2) copies n characters from s1, starting at s1[p1], to s2, starting at s2[p2]. String length is updated, if needed, to p2 + (n − 1) if p2 + (n-1) > strlen(s2).

If p2 equals `strlen`(s2) + 1, `strmove` is equivalent to appending a subset of s1 to s2.

You may use `strmove` to convert PAC's to strings and vice versa. It is also an efficient way of manipulating subsets of PAC's.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

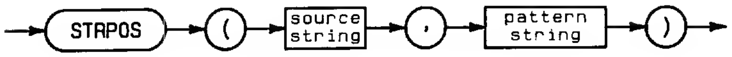
You should not strmove into an uninitialized variable regardless of its type.

Example Code

```
VAR
  pac: PACKED ARRAY[1..15] OF char;
    s: string[80];
BEGIN
   s:= '';
   pac:= 'Hewlett-Packard';
   strmove(15,pac,1,s,1);    {Converts a PAC to a string.}
END.
```

strpos

This function returns the starting position of the first occurrence of a series of characters within a string.



Item	Description/Default	Range Restrictions
source string	expression of type string	-
pattern string	expression of type string	-

Example

```
strpos(source,pattern)
```

Semantics

The function `strpos`(s1,s2) returns the integer index of the position of the first occurrence of s2 in s1. If s2 is not found, zero is returned.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Note

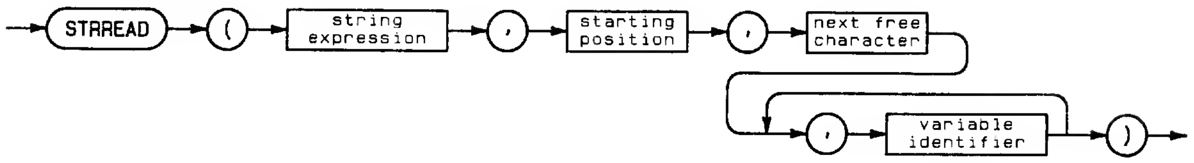
Some HP Pascal implementations have the order of the two parameters reversed. Also, a compiler option may exist for reversing the order of parameters.

Example Code

```
CONST
  separator = ' ';
VAR
  i: integer;
  names: string[80];
BEGIN
  .
  names:= 'Jon Jill Ruth Marnie Bob Joan Wendy';
  i:= strpos (names,separator);
  IF i <> 0 THEN
    strdelete(names,1,i);          {deletes first name}
  .
END
```

strread

This procedure reads a value from a string as if it were an external textfile.



Item	Description/Default	Range Restrictions
string expression	expression of type string	-
starting position	expression of type integer	-
next free character	variable of an integer or integer subrange type	-
variable identifier	simple, string, or PAC variable	-

Examples

```
strread(str_exp,start_pos,next_char,variable)
strread(str_exp,start_pos,next_char,variable1,...,variablen)
```

Semantics

The procedure `strread`(s,p,t,v) reads a value from s, starting at s[p], into the variable v. After the operation, the value of the variable appearing as the t parameter will be the index of s immediately after the index of the last component read into v.

S is treated as a single-line textfile. `Strread`(s,p,t,v) is analogous to `read`(f,v) when f is a textfile of one line. Like `read`, `strread` implicitly converts a sequence of characters from s into the types `integer`, `real`, `longreal`, `boolean`, `enumerated`, PAC, or `string`.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

An error occurs if `strread` attempts to read beyond the current length of s.

The call

```
strread(s,p,t,v1,...,vn);
```

is equivalent to

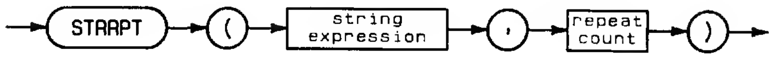
```
strread(s,p,t,v1);
strread(s,t,t,v2);
  .
  .
strread(s,t,t,vn);
```

Example Code

```
VAR
  s: string[80];
  p,t: 1..80;
  m,n: integer;
BEGIN
   .
  s:= '   12  564    ';
   .
  p:= 1;
  strread(s,p,t,m);      {The value of m will be 12; }
   .                     {t will be 6.              }
   .
  strread(s,t,t,n);      {The value of n will be 564;}
   .                     {t will be 11.             }
END.
```

strrpt

This function returns a string composed several copies of its string argument.



Item	Description/Default	Range Restrictions
string expression	expression of type string	-
repeat count	expression of type integer	-

Example

```
strrpt(str_exp,rep_count)
```

Semantics

The function `strrpt(s,n)` returns a string composed of s repeated n times.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

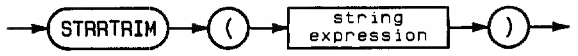
Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

```
CONST
  one = '1';
VAR
  b_num: string[32];
BEGIN
  .
 b_num:= strrpt(one,strmax(b_num));
  .
END.
```

strrtrim

This function returns a string trimmed of trailing blanks.



Example

```
strrtrim(str_exp)
```

Semantics

The function `strrtrim(s)` returns a string consisting of s trimmed of trailing blanks. Leading blanks are stripped by the function `strltrim` (see above).

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

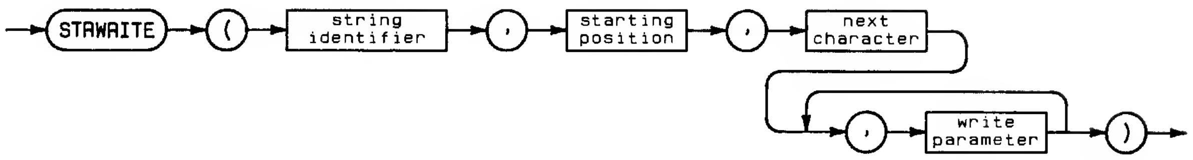
Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

Example Code

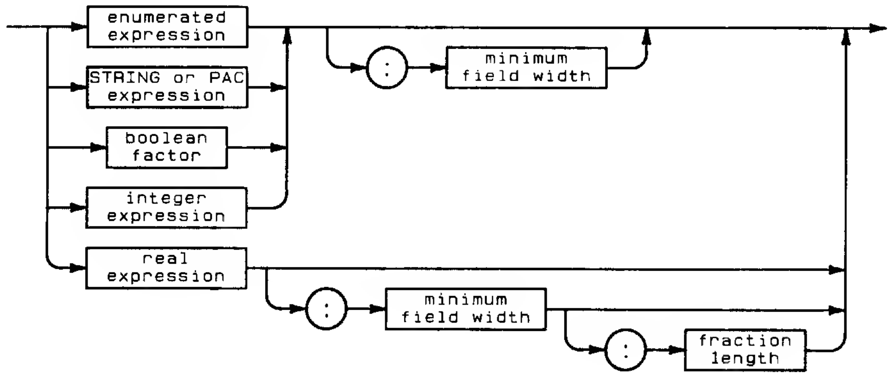
```
VAR
  s: string[80]
BEGIN
  .
  s:= 'abc         ';
  .
  s:= strrtrim(s);         {s is now 'abc'}
  .                            {strlen(s) = 3 }
END.
```

strwrite

This procedure writes a value to a string as if it were an external textfile.



Write Parameter



Item	Description/Default	Range Restrictions
string identifier	variable of type string	-
starting position	expression of type integer	1 thru current length of the string + 1
next character	variable of an integer or integer subrange type	-
write parameter	see drawing	-
minimum field width	integer expression	greater than 0
fraction length	integer expression	greater than 0

Examples

```
strwrite(str_exp,start_pos,next_char,variable)
strwrite(str_exp,start_pos,next_char,variable1,...,variablen)
```

Semantics

The procedure strwrite(s,p,t,e) writes the value of e on s starting at s[p]. After the operation, the value of the variable appearing as the t parameter will be the index of the component of s immediately after the last component of s that strwrite has accessed.

S is treated as a single-line textfile. Strwrite(s,p,t,e) is analogous to write(f,e) when f is a one-line textfile. As with write, strwrite also permits you to format the value of e as it is written to s using the formatting conventions. The same default formatting values hold for strwrite.

Strwrite may write into the middle of a string without affecting the original length.

An error occurs if strwrite attempts to write beyond the maximum length of s, or if p is greater than strlen(s) + 1.

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Strings must be initialized just like any other variable. The string functions and procedures assume that the string parameters contain valid information.

The call

```
strwrite(s,p,t,e1,..,en);
```

is equivalent to

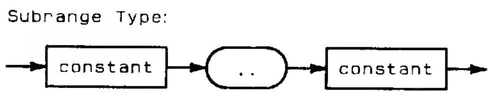
```
 strwrite(s,p,t,e1);
 strwrite(s,t,t,e2);
   .
   .
 strwrite(s,t,t,en);
```

Example Code

```
VAR
  s: string[80]
  p,t: 1..80;
  f,g: integer;
BEGIN
  f:= 100;
  g:= 99;
  p:=1;
   .
  strwrite(s,p,t,f:1);       {S is now '100'; t is 4      }
  strwrite(s,t,t,' ',g:1);   {S is now '100 99'; t is 7. }
   .
END.
```

Subrange

A subrange type is a sequential subset of an ordinal host type. A subrange type consists of a lower bound and an upper bound separated by the special symbol ".." (i. e. 10..99). The upper and lower bounds must be constant values of the same ordinal type and the lower bound cannot be greater than the upper bound.



A constant expression may appear as an upper or lower bound.

A subrange type is a simple ordinal type: boolean, char, integer, and user-defined enumeration or subrange types.

Permissible Operations and Standard Functions

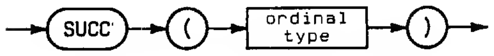
A variable of a subrange type possesses all the attributes of the host type of the subrange, but its values are restricted to the specified closed range.

Example Code

```
TYPE
  day_of_year = 1..366;

  lowercase   = 'a'..'z';          {Host type is char.    }

  days        = (Monday, Tuesday, Wednesday,
                   Thursday,Friday,Saturday,Sunday);
  weekdays    = Monday..Friday;
  weekend     = Saturday..Sunday;

  e_type      = 1..maxsize - 1     {Upper bound is con-   }
                                   {stant expression.     }
                                   {Maxsize is declared   }
                                   {constant.             }
```

succ

This function returns the value whose ordinal number is one greater than the ordinal number of the argument.



Examples

Input	Result
`succ(ord_type)`	
`succ(1)`	`2`
`succ(-5)`	`-4`
`succ('a')`	`'b'`
`succ(false)`	`true`
`succ(true)`	`{error}`

Semantics

The function `succ`(x) returns the value, if any, whose ordinal number is one greater than the ordinal number of x. The type of the result is identical with the type of x. A run-time error occurs if `succ`(x) does not exist. For example, suppose:

```
   TYPE color = (red, blue, yellow)
```

`Then,`

```
   succ(red) = blue
```

but `succ`(yellow) is undefined.

Symbols

The following table lists the special symbols valid in HP Pascal.

Symbol	Purpose
+	add, set union, concatenate strings
-	subtract, set difference
*	multiply, set intersection
/	divide (real results)
=	equal to
<	less than
>	greater than
( )	delimit a parameter list or a subexpression
[ ]	delimit an array index or a constructor. May be replaced by (. or .)
.	select record field, decimal point
,	separate listed identifiers
;	delimit statements
:	delimit list of identifiers
^	define or dereference pointers, access file buffer. May be replaced by @.
<>	not equal
<=	less than or equal, subset
>=	greater than or equal, superset
:=	assign value to a variable
..	subrange
{ }	delimit a comment. May be replaced by (* or *)
#	encode a control character
$	delimit a compiler option
'	delimit a string literal
_	may appear within an identifier

Separators may not appear within special symbols having more than one component (e.g. :=).

Certain special symbols have synonyms. In particular, (. and .) may replace the left and right brackets [and]. The symbol @ may substitute for the up-arrow ^, also (* and *) may take the place of the left and right braces, { and }.

text

The standard file type `text` permits ordinary input and output oriented to characters and lines. `Text` type files have two important features:

1. The components are type `char`.
2. The file is subdivided into lines by special end-of-line markers.

`Text` type variables are called "textfiles".

A text file type consists of the predefined type `text`.

Textfiles cannot be opened for direct access with the procedure `open`. Textfiles may be sequentially accessed, however, with the procedures `reset`, `rewrite`, or `append`. All standard procedures that are legal for sequentially accessed files are also legal for textfiles.

Certain standard procedures and functions, on the other hand, are legal only for textfiles: `readln`, `writeln`, `page`, `prompt`, `overprint`, `eoln`, and `linepos`.

Textfiles permit conversion from the internal form of certain types to an ASCII character representation and vice versa.

Example Code

```
VAR
  myfile: text;
```

THEN

See IF.

TO

See FOR.

true

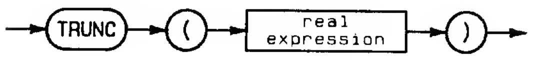
This predefined constant is equal to the boolean type whose value is true.

Example Code

```
PROGRAM show_true(output);

TYPE
  what, truth : boolean;

BEGIN
  IF true THEN writeln('always true, always printed');
  what := true;
  truth := NOT false;
  IF what = truth THEN writeln('Everything I say is a lie.');
END.
```

trunc

This function returns the integer part of a real or longreal expression.



Examples

Input	Result
`trunc(real_exp)`	
`trunc(5.61)`	5
`trunc(-3.38)`	-3
`trunc(18.999)`	18

Semantics

The function `trunc(x)` returns an integer result which is the integral part of x. The absolute value of the result is not greater than the absolute value of x. An integer overflow occurs if the result is not in the range `minint..maxint`.

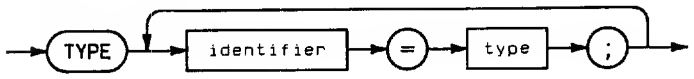
TYPE

This reserved word delimits the start of the type declarations in a program, module, procedure or function.

A type definition establishes an identifier as a synonym for a data type. The identifier may then appear in subsequent type or constant definitions, or in variable declarations.

The reserved word TYPE precedes one or more type definitions. A type definition consists of an identifier, the equals sign (=), and a data type.

Type Definition:



A data type determines a set of attributes which include:

- the set of permissible values
- the set of permissible operations
- the amount of storage required

Subsequent pages explain the permissible values and operations for the various data types.

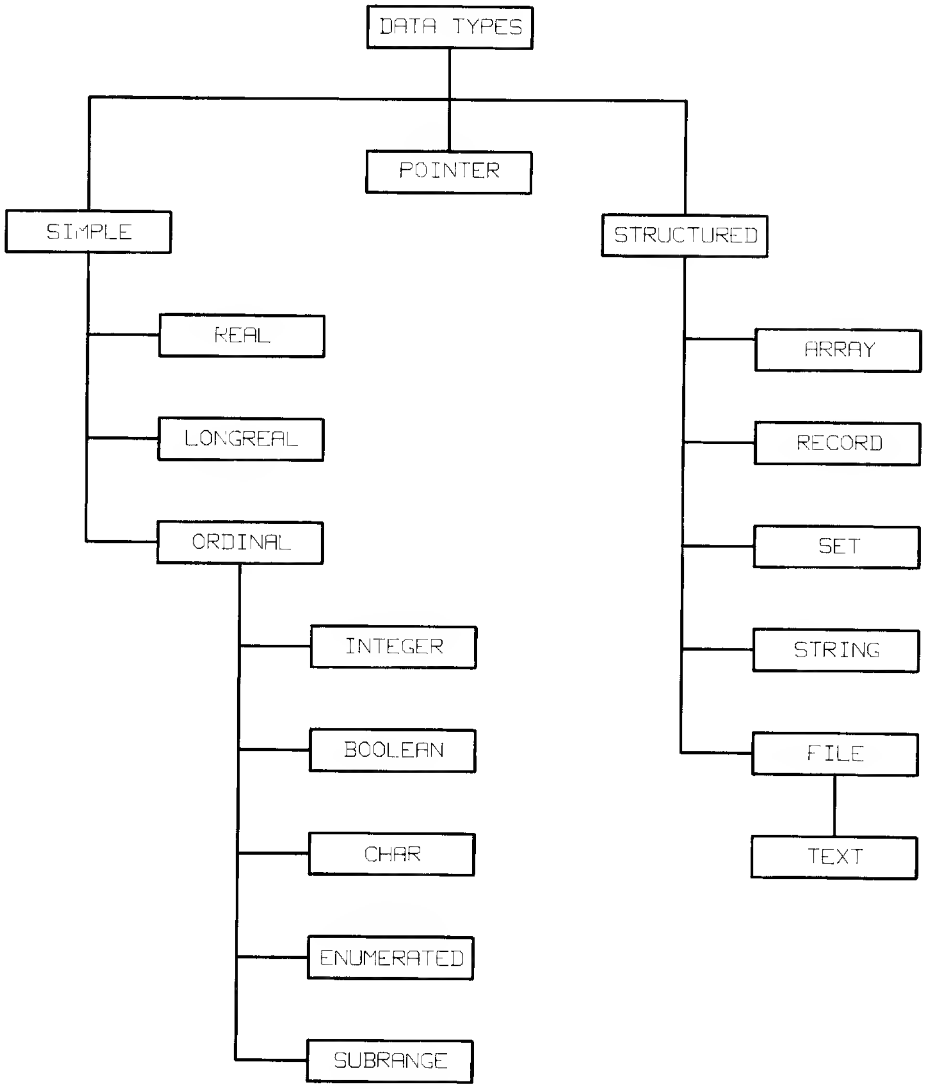
The three most general categories of data type are simple, structured, and pointer.

Simple data types are the types ordinal, `real`, or `longreal`. Ordinal types include the standard types `integer`, `char`, and `boolean`, as well as user-defined enumerated and subrange types.

Structured data types are the types array, record, set, or file. The standard type `string` is also a structured data type. The standard type `text` is a variant of the file type.

Pointer data types define pointer variables which point to dynamically allocated variables on the heap.

The following figure shows the relation of these various categories.



HP Pascal Data Types

Type Compatibility

Relative to each other, two HP Pascal types can be identical, type compatible, or incompatible.

Identical Types

Two types are identical if either of the following is true:

1. Their types have the same type identifier.
2. If A and B are their two type identifiers, and they have been made equivalent by a definition of the form:

```
TYPE A = B
```

Compatible Types

Two types T1 and T2 are type compatible if any of the following is true.

1. T1 and T2 are identical types.
2. T1 and T2 are subranges of the same host type, or T1 is a subrange of T2, or T2 is a subrange of T1.
3. T1 and T2 are set types with compatible base types and both T1 and T2 or neither are packed.
4. T1 and T2 are PAC types with the same number of components, or if either T1 or T2 is a character constant or a string literal constant whose length is less than the length of the other type, in which case the constant is extended on the right with blanks to reach a compatible length.
5. T1 and T2 are both `string` types.
6. T1 and T2 are both real types, i.e. `real` or `longreal`.

Incompatible Types

Two types are incompatible if they are not identical, type compatible, or assignment compatible.

Example Code

```
TYPE
   interval = 0..10;
   range = interval;

VAR
   v1 : 0..10;
    v2, v3: 0..10;
   v4 : interval;
   v5 : interval;
   v6 : range;
```

All of the variables are type compatible, but v4, v5, and v6, have identical types. The variables v2 and v3 also have identical types.

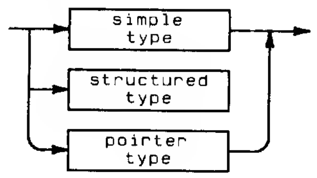
Just because two types look compatible, it does not mean they are compatible. In the following example, type T1 and T2 are **not** compatible.

```
TYPE
   T1 = record
           a : integer;
           b : char;
        end;

  T2 = record
           c : integer;
           d : char;
        end;
```

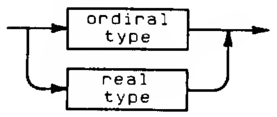
Types

The following data types are available in HP Pascal.

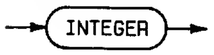
Type:



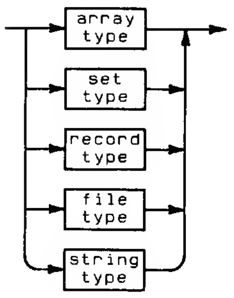
Simple Type:



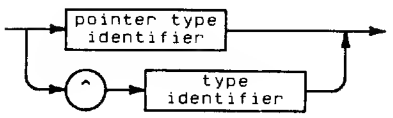
Integer Type:



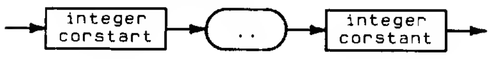
Structured Type:



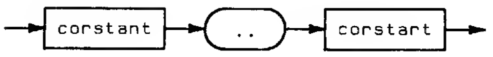
Pointer Type:



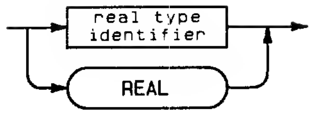
Integer Subrange Type



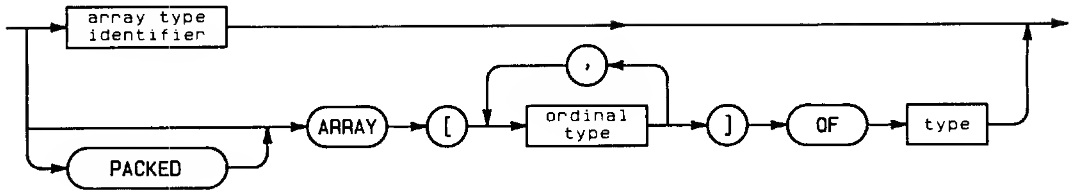
Subrange Type:



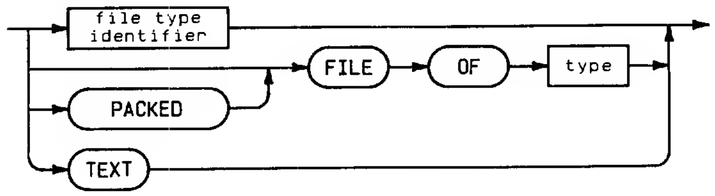
Real Type:



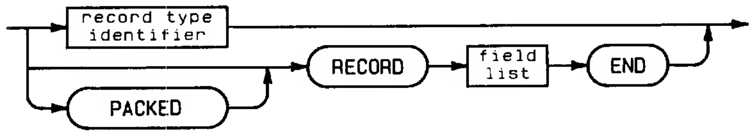
Array Type:



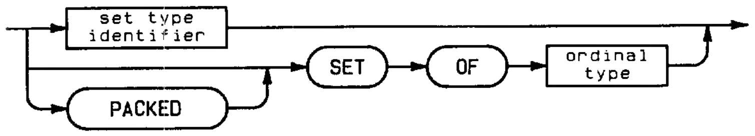
File Type:



Record Type:

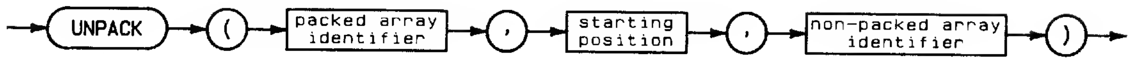


Set Type:



unpack

This procedure transfers data from a packed array to a regular array.



Item	Description/Default	Range Restrictions
packed array identifier	variable of type array	see semantics
starting position	expression which is type compatible with the index of the non-packed array	-
non-packed array identifier	variable of type PACKED array	see semantics

Example

```
unpack(packed_array,start_pos,array)
```

Semantics

Assuming `a : ARRAY[m..n] OF t` and `x : PACKED ARRAY [u..v] OF t`; the procedure `unpack(z,i,a)` successively assigns the components of the packed array z, starting at component u, to the components of the unpacked array a, starting at a[i].

All the components of z are assigned. Hence, z must be shorter than or as long as a, i.e. $(v - u) <= (n - m)$. Also, the normalized value of i must be less than or equal to the difference between the lengths of a and z plus 1, i.e. $i - m + 1 <= (n - m) - (v - u) + 1$. Otherwise, an error occurs when `unpack` attempts to index a beyond its upper bound (see example below).

The index types of a and z need not be compatible. The components of the two arrays, however, must be type identical.

The call `unpack(z,i,a)` is equivalent to:

```
BEGIN
   k:= i;
  FOR j:= u TO v DO
     BEGIN
      a[k]:= z[j];
      IF j <> v THEN k:= succ(k);
    END;
 END;
```

where k and j are variables that are type compatible with the indices of a and z respectively.

Example Code

```
PROGRAM show_unpack (input,output);
TYPE
  suit_types = (casual, business, leisure, birthday);
VAR
   suit : PACKED ARRAY [1..5] OF suit_types;
  kase : ARRAY [1..10] OF suit_types;
    .
    .
BEGIN
    .
    .
    unpack(suit,1,kase);  {After execution, the first 5      }
    .                     {components of kase contain the    }
    .                     {value of suit.                    }
    .
    unpack(suit,7,kase);  {An error results because unpack   }
    .                     {attempts to assign a component of }
    .                     {suit to a component of kase which }
    .                     {is out of range.                  }
END.
```

UNTIL

See REPEAT.

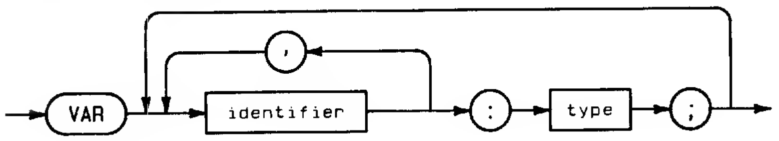
VAR

This reserved word delimits the beginning of variable declarations in a Pascal program or module.

A variable declaration associates an identifier with a type. The identifier may then appear as a variable in executable statements.

The reserved word VAR precedes one or more variable declarations. A variable declaration consists of an identifier, a colon (:), and a type. Any number of identifiers may be listed separated by commas. These identifiers will then be variables of the same type.

Variable Declaration:



The type may be any simple, structured, or pointer type. The form of the type may be a standard identifier, a declared type identifier, or a data type (see example below).

You may repeat VAR sections and intermix them with CONST and TYPE sections.

Components of a structured variable may be accessed using an appropriate selector. Pointer variable dereferencing accesses dynamic variables on the heap.

HP Pascal predefines two standard variables, `input` and `output`, which are textfiles. Formally,

```
VAR
  input, output: text;
```

These standard textfiles commonly appear as program parameters and serve as default files for various file operations.

Each variable is a statically declared object and is accessible for the duration of the program procedure or function in which it is declared. Module variables are accessible for the duration of the program which imports the module.

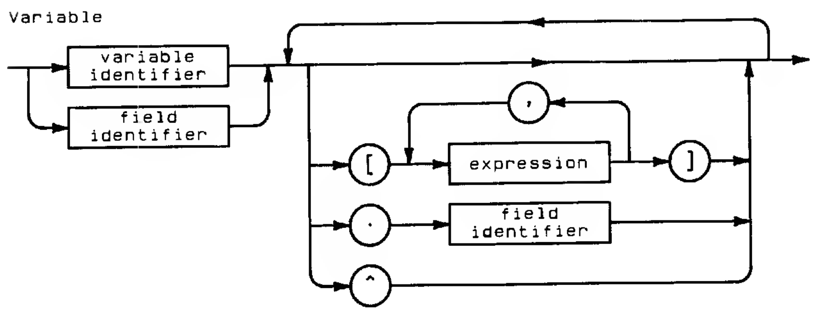
Every declaration of a file variable F with components of type T implies the additional declaration of a buffer variable of type T. The buffer variable, denoted as F^, may be used to access the current component of the file F.

Example Code

```
TYPE
  answer = (yes, no, maybe);
VAR
  pagecount,
  linecount,
  charcount: integer;           {Standard identifier.     }

  whats_the: answer;            {User-declared identifier.}

  album    : RECORD             {Data type.               }
                speed: (lp, for5, sev8);
                price: real;
                name : string[20];
             END;
```

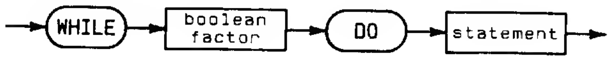
Variables

A variable appearing in an executable statement takes the following form.



WHILE

The WHILE statement executes a statement repeatedly as long as a given condition is true. The WHILE statement consists of the reserved word WHILE, a boolean factor (the condition), the reserved word DO, and a statement.



When the system executes a WHILE statement, it first evaluates the condition. If the condition is true, it executes the statement after DO and then re-evaluates the condition. When the condition becomes false, execution resumes at the statement after the WHILE statement. If the condition is false at the beginning, the system never executes the statement after DO.

The statement

```
WHILE condition DO statement
```

is equivalent to:

```
1: IF condition THEN BEGIN
      statement;
      GOTO 1;
   END;
```

Usually a program will modify data at some point so that the condition becomes false. Otherwise, the statement will repeat indefinitely. It is also possible, of course, to branch unconditionally out of a WHILE statement using a GOTO statement.

The compiler can be directed to perform partial evaluation of boolean operators used in WHILE statements. For example:

```
WHILE a_one AND a_two DO ...
```

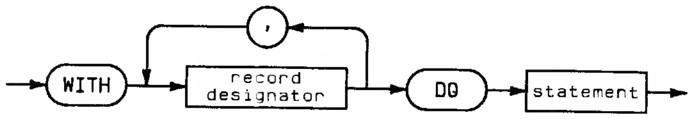
By specifying the $PARTIAL_EVAL ON$ compiler directive, if "a_one" is `false`, the remaining operators will not be evaluated since execution of the statement depends on the logical AND of both operators. (Both operators would have to be `true` for the logical AND of the operators to be `true`.)

Example Code

```
WHILE index <= limit DO
  BEGIN
    writeln (real_array [index]);
    index := index + 1;
  END;
  .
  .
WHILE NOT eof (f) DO
  BEGIN
    read (f, ch);
    writeln (ch);
  END;
```

WITH

A WITH statement allows you to refer to record fields by field name alone. A WITH statement consists of the reserved word WITH, one or more record designators, the reserved word DO, and a statement.



A **record designator** may be a record identifier, a function call which returns a record, or a selected record component.

The statement after DO may be a compound statement. In this statement, you can refer to a record field contained in one of the designated records without mention of the record to which it belongs. The appearance of a function reference as a record designator is an invocation of the function.

You may not assign a new value to a field of a record constant or a field of a record returned by a function.

When the system executes a WITH statement, it evaluates the record designators and then executes the statement after DO.

The following statements are equivalent:

```
WITH rec DO
  BEGIN
    field1 := e1;
    writeln(field1 * field2);
  END;
```

```
BEGIN
  rec.field1 := e1;
  writeln(rec.field1
          * rec.field2);
END;
```

Since the system evaluates a record designator once and only once before it executes the statement, the statement sequence, where f is a field,

```
i := 1;
WITH a[i] DO
  BEGIN
    writeln(f);
    i:=2;
    writeln(f)
  END;
```

produces the same effect as:

```
writeln(a[1].f);
writeln(a[2].f);
```

Records with identical field names may appear in the same WITH statement. The following interpretation resolves any ambiguity:

The statement

```
WITH record1, record2, ..., recordn DO
  BEGIN
    statement;
  END;
```

is equivalent to

```
WITH record1 DO
  BEGIN
    WITH record2 DO
      BEGIN
        ...
         WITH recordn DO
           BEGIN
             statement;
           END;
        ...
      END;
  END;
```

Thus, if field f is a component of both record1 and record2, the compiler interprets an unselected reference to f as a reference to record2.f. You may access the synonymous field in record1 using normal field selection, i.e. record1.f.

This interpretation also means that if r and f are records, and f is a field of r, then the statement

```
WITH r DO
  BEGIN
    WITH r.f DO
      BEGIN
        statement;
      END;
  END;
```

is equivalent to

```
WITH r.f DO
  BEGIN
    statement;
  END;
```

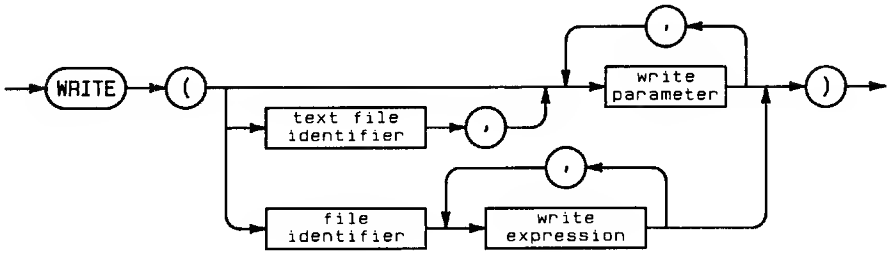
If a local or global identifier has the same name as a field of a designated record in a WITH statement, then the appearance of the identifier in the statement after DO is always a reference to the record field. The local or global identifier is inaccessible.

Example Code

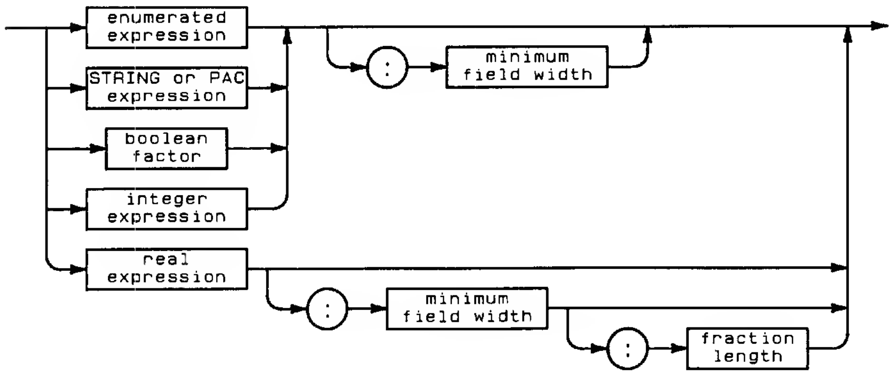
```
PROGRAM show_with;

TYPE
  status = (married, widowed, divorced, single);
  date   = RECORD
             month  : (jan, feb, mar, apr, may, jun,
                       july, aug, sept, oct, nov, dec);
             day    : 1..31;
             year   : integer;
           END;
  person = RECORD
             name   : RECORD
                        first, last: string[10]
                      END;
             ss     : integer;
             sex    : (male, female);
             birth  : date;
             ms     : status;
             salary : real
           END;

VAR
  employee : person;

BEGIN {show_with}
  .
  WITH employee, name, birth DO
    BEGIN
      last := 'Hacker';
      first := 'Harry';
      ss := 2147483647;
      sex := male;
      month := feb;
      day := 29;
      year := 1952;
      ms := single;
      salary := 32767.0
    END;
  .
END {show_with}
```

write

This procedure assigns a value to the current component of a file and then advances the current position.



Write Parameter



Item	Description/Default	Range Restrictions
textfile identifier	file of type text; defaults = output	file must be opened
write parameter	see drawing	-
file identifier	variable of type file	must be opened to write
write expression	expression	must be type compatible with file
minimum field width	integer expression	greater than 0
fraction length	integer expression	greater than 0

Examples

```
write(file_var,exp:5)
write(file_var,exp1,...,expn)
write(exp)
write(exp1,...,expn)
```

Semantics

The procedure `write`(f,e) assigns the value of e to the current component of f and then advances the current position. After the call to `write`, the buffer variable f^ is undefined. An error occurs if f is not open in the write-only or read-write state. An error also occurs if the current position of a direct access file is greater than `maxpos`(f).

If **f** is not a textfile, an expression whose result type is assignment compatible with the components of **f**. If **f** is a textfile, e may be an expression whose result type is any simple or `string` type, a variable of type `string` or PAC, or a string literal. Also, you may format the value of **e** as it is written to a textfile (see below).

The call `write(f,e)` is equivalent to

```
f^ := e;
put(f);
```

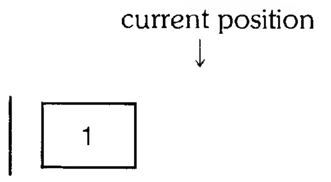
The call `write`(f,e1,...en) is equivalent to

```
write(f,e1);
write(f,e2);
  .
  .
write(f,en);
```

Illustration

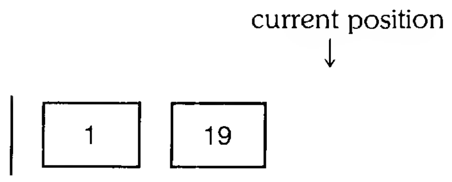
Suppose examp_file is a file of `integer` opened in the write-only state and that we have written one number to it. To write another number, we call `write` again:

{initial condition}



state: write-only
examp_file^: undefined
eof(examp_file): true

write(examp_file,19);



state: write-only
examp_file^: undefined
eof(examp_file): true

Formatting of Output to Textfiles

When f is a textfile, the result type of e need not be `char`. It may be any simple, `string`, or PAC type, or a string literal. The value of e may be formatted as it is written to f using the integer field-width parameters m and, for real or longreal values, n. If m and n are omitted, the system uses default formatting values. Thus, three forms of e are possible in source code:

```
e       {default formatting}
e:m     {when e is any type}
e:m:n   {when e is real or longreal}
```

The following table shows the system default values for m.

Default Field Widths

Type of e	Default Field Width (m)
char	1
integer	12
real	13
longreal	22
boolean	length of identifier
enumerated	length of identifier
string	current length of string
PAC	length of PAC
string literal	length of string literal

If e is `boolean` or an enumerated type, what gets written is implementation defined.

When m is specified and the value of e requires less than m characters for its representation, the operation writes e on f preceded by an appropriate number of blanks. If the value of e is longer than m, it is written on f without loss of significance, i.e. m is defeated, provided that e is a numeric type. Otherwise, the operation writes only the leftmost m characters. M may be 0 if e is not a numeric type.

When e is type `real` or `longreal`, you may specify n as well as m. In this case, the operation writes e in fixed-point format with n digits after the decimal point. If n is 0, the decimal point and subsequent digits are omitted. If you do not specify n, the operation writes e in floating-point format consisting of a coefficient and a scale factor. In no case is it possible to write more significant digits than the internal representation contains. This means `write` may change a fixed-point format to a floating-point format in certain circumstances.

Example Code

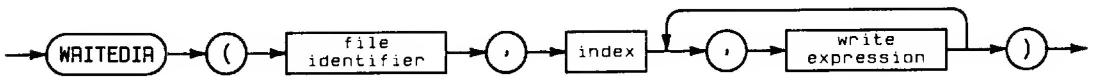
```
PROGRAM show_formats (output);
VAR
  x: real;
  lr: longreal;
  george: boolean;
  list: (yes, no, maybe);
BEGIN
  writeln(999);              {default formatting}
  writeln(999:1);            {format defeated}
  writeln('abc');
  writeln('abc':2);          {string literal truncated}
  x:= 10.999;
  writeln(x);                {default formatting}
  writeln(x:25);
  writeln(x:25:5);
  writeln(x:25:1);
  writeln(x:25:0);
  lr:= 19.1111;
  writeln(lr);
  george:= true;
  writeln(george);           {default format}
  writeln(george:2);
  list:= maybe;
  write(list);               {default formatting}
END.
```

The output of this program is:

```
         999
999
abc
ab
 1.099900E+01
             1.099900E+01
                 10.99900
                     11.0
                       11
 1.91110992431641L+001
TRUE
TR
MAYBE
```

writedir

This procedure places the current position at the specified component and then writes the value of its argument to that component.



Item	Description/Default	Range Restrictions
file identifier	variable of type file	file must be open to write; file must not be a textfile
index	integer expression	greater than 0; less than `lasPos`(file identifier)
write expression	expression that is type compatible with file type	see semantics

Examples

```
writedir(fil_var,indx,exp)
writedir(fil_var,indx,exp1,...,expn)
```

Semantics

The procedure `writedir`(f,k,e) places the current position at the component of f specified by k and then writes the value of e to that component. It is equivalent to

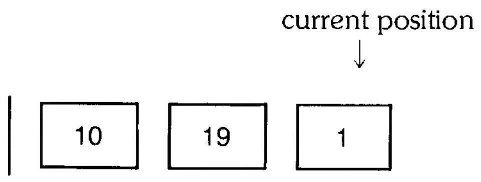
```
seek(f,k);
write(f,e)
```

An error occurs if f has not been opened in the read-write state or if k is greater than `maxpos`(f). After `writedir` executes, the buffer variable f^ is undefined and the current position is k + 1.

Illustration

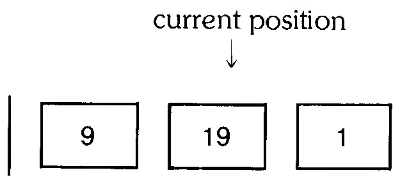
Suppose file examp_file is a file of `integer` opened for direct access. The current position is the third component. To write a number to the first component, we call `writedir`:

{initial condition}



state: read-write
examp_file^(deferred): 1
eof(examp_file): false

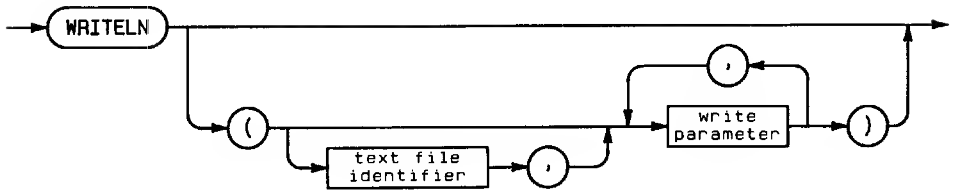
writedir(examp_file,1,4 + 5);



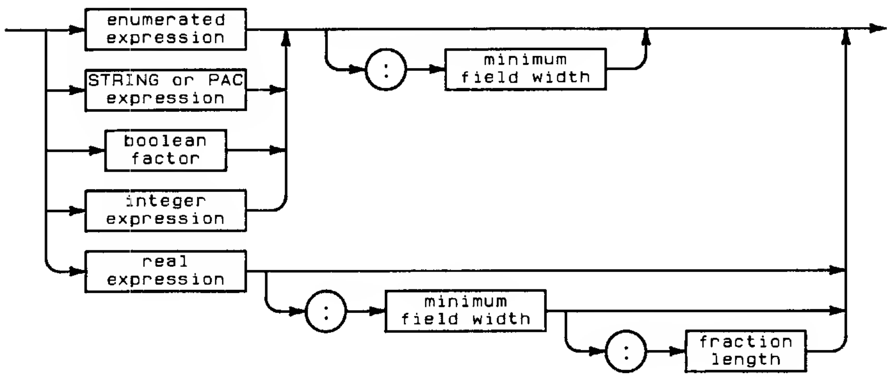
state: read-write
examp_file^: undefined
eof(examp_file): false

writeln

This procedure writes the value of its argument to a textfile.



Write Parameter



Item	Description/Default	Range Restrictions
textfile identifier	file of type text; default = `output`	file must be opened to write
write parameter	see drawing	-
minimum field width	integer expression	greater than 0
fraction length	integer expression	greater than 0

Examples

```
writeln(fil_var)
writeln(fil_var,exp:4)
writeln(fil_var,exp1,...,expn)
writeln(exp)
writeln(exp1,...,expn)
writeln
```

Table of Contents

Semantics

The procedure `writeln`(f,e) writes the value of the expression e to the textfile f, appends an end-of-line marker, and places the current position immediately after this marker. After execution, the file buffer f^ is undefined and `eof`(f) is `true`. You may write the value of e with the formatting conventions described for the procedure `write`.

The call `writeln`(f,e1,...,en) is equivalent to

```
write(f,e1);
write(f,e2);
   .
   .
   .
write(f,en);
writeln(f)
```

The call `writeln` without the file or expression parameters effectively inserts an empty line in the standard file `output`.

Implementation Appendix

Series 200 HP-UX

This appendix describes the implementation-specific details of HP Pascal for the HP-UX operating system on the Series 200 Computers.

The following topics are described in this appendix.

- Compiler Options
- Implementation Dependencies
- Replacements for Pascal Extensions
- System Programming Language Extensions
- Special Use of RESET and REWRITE
- Unbuffered Terminal Input
- The HP-UX **pc** Command
- Program Parameters and Program Arguments
- Pascal Heap Managers for Series 200
- Using Pascal with other Languages
- Pascal Run-Time Error Handling
- Error Messages

Compiler Options

The pages in this section describe the compiler options (compiler directives) you may use with Pascal on Series 200 HP-UX systems. When specified, compiler options usually have a default action and restrictions on where they may appear. These restrictions are shown on every page below the option.

The explanation of these restrictions is given below.

Restrictions on the Placement of Compiler Directives

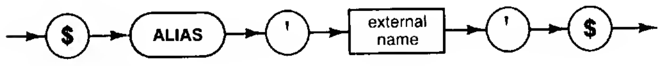
Location	**Restriction**
Anywhere:	No restriction.
At front:	Applies to entire source file; must appear before the first "token" in the source file (before PROGRAM, or before MODULE if compiling a list of modules).
Not in body:	Applies to a whole procedure or function; can't appear between BEGIN and END. Good practice to put these options immediately before the word BEGIN, or the procedure heading.
Statement:	Can be applied on a statement-by-statement basis or to a group of statements, by enabling before and disabling after the statements of interest.
Special:	As explained under the particular option.

If a option appears in the interface (import or export) part of a module, it will have effect as the module is compiled. However, the option itself will not become part of the interface specification (export text) in the compiled module's object code and will have no effect in the implement section of the module being compiled.

ALIAS

Default: External name = Procedure Name
Location: Special, See Below

This option causes a name, other than the name used in the Pascal procedure or function declaration, to be used by the loader.



Item	Description/Default	Range Restrictions
external name	string	Entire declaration must fit on one line.

Semantics

The string parameter specifies the external name for the procedure in whose header the option appears.

Example

```
procedure $alias 'charlie'$ p (i: integer);  external;
```

Within the program, calls use the name "p"; but the loader will link to a physical routine called "charlie".

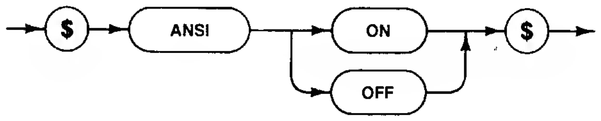
The option must appear between the keywords PROCEDURE or FUNCTION and the first symbol following the semicolon (;) denoting the end of the procedure or function declaration.

The option may not appear in an export section.

ANSI

Default: OFF
Location: At Front

This option selects whether an error message is to be emitted for use of any feature of HP Standard Pascal not contained in ANSI/ISO Standard Pascal.



Semantics

"ANSI" is interpreted as "ANSI ON".

ON causes error messages to be issued for use of any feature of HP Standard Pascal which is not part of ANSI/ISO Standard Pascal.

OFF suppresses the error messages.

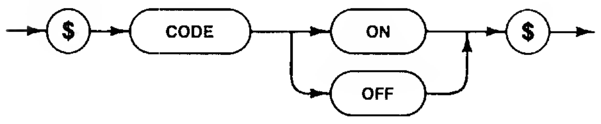
Example

```
$ansi on$
```

CODE

Default: ON
Location: Not in Body

This option is used to control whether a CODE file will be generated by the compiler.



Semantics

"CODE" is interpreted as "CODE ON".

ON specifies that executable code will be emitted.

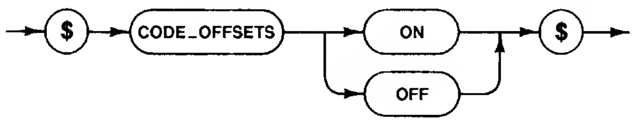
Example

```
$code off$
```

CODE_OFFSETS

Default: OFF
Location: Not in Body

This option controls the inclusion of program counter offsets in the compiler listing.



Semantics

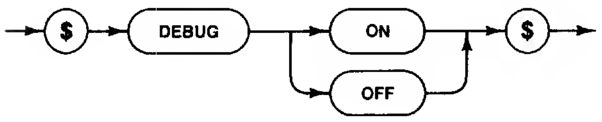
"CODE_OFFSETS" is interpreted as "CODE_OFFSETS ON".

ON specifies that line number-program counter pairs will be printed for each executable statement listed. This can be applied on a procedure-by-procedure basis.

DEBUG

Default: OFF
Location: Not in Body

This option controls whether the code produced by the compiler contains the additional information necessary for reporting line number information with error messages.



Semantics

"DEBUG" is interpreted as "DEBUG ON"

"DEBUG ON" will cause instructions to be emitted, which assign the current line number to the system variable "asm_line", for the procedure bodies following it. These instructions are not stripped by the *strip*(1) command of HP-UX.

This option may be applied on a procedure-by-procedure basis.

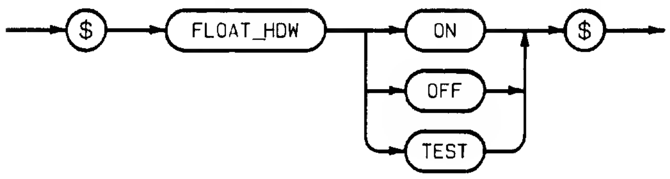
Example

```
procedure buggy;
var  i: integer;
$debug on$
begin
  ...
end;
$debug off$
```

FLOAT_HDW

Default: OFF
Location: Not in body

This option enables and disables the use of floating-point hardware.



Semantics

An optional floating-point hardware board is available for Series 200 Computers to increase the execution speed of floating-point math programs.

A small overhead occurs on every procedure when this option is enabled. For maximum performance, bracket calls to math-intensive procedures with `$FLOAT ON$` and `$FLOAT OFF$`.

"FLOAT_HDW" is interpreted as "FLOAT_HDW ON"

ON instructs the compiler to generate accesses to hardware for most floating-point operations. If the hardware does not exist when the program is executed, an error will result.

OFF tells the compiler to generate calls to libraries for all floating-point operations.

TEST causes the compiler to generate both hardware accesses and library calls. The compiler automatically includes code to test for the presence of floating-point hardware. At execution time, if the test succeeds, the hardware accesses are used, otherwise the library calls are used.

The operations that use the hardware include: addition, subtraction, multiplication, division, negation, and the `sqr` function. All other math functions call library routines. There are libraries that access the floating-point hardware. Hardware can also be used by any operation that converts an integer to a real or longreal, converts a real to a longreal, or converts a longreal to a real. The hardware is not used by operations that convert reals or longreals into integers.

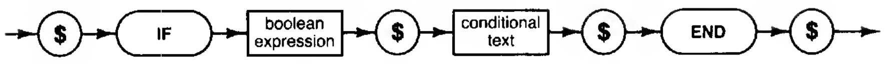
Example

```
$float test$
```

IF

Default: Not Applicable
Location: Anywhere

This option allows conditional compilation.



Item	Description/Default	Range Restrictions
boolean expression	-	may only contain compile time constants
conditional text	source to be conditionally compiled	

Semantics

If the expression evaluates to FALSE, then text following the option is skipped up to the next END option.

If the boolean evaluates to TRUE, then the text following the option is compiled normally.

IF-END option blocks may not be nested.

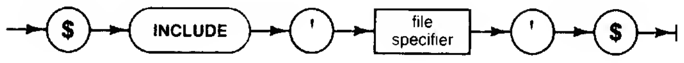
Example

```
const  fancy = true;
       limit = 10;
       size = 9;
...
$if fancy and ((size+1)<limit)$
  ...   (* this will be skipped *)
$end$
```

INCLUDE

Default: Not Applicable
Location: Anywhere

This option allows text from another file to be included in the compilation process.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier

Semantics

The string parameter names a file which contains text to be included at the current position in the program. Included code may contain additional INCLUDE options.

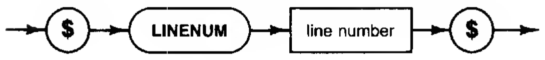
Example

```
program inclusive;
$include '/users/steve/declars'$
$include '/users/steve/body'$
end.
```

LINENUM

Default: Not Applicable
Location: Anywhere

This option allows the user to establish an arbitrary line number value.



Item	Description/Default	Range Restrictions
line number	integer numeric constant	1 thru 65534

Semantics

The integer parameter becomes the current line number (for listing purposes and debugging purposes if $debug$ is enabled).

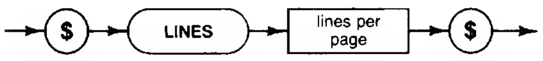
Example

```
$linenum 20000$
```

LINES

Default: 60 lines per page
Location: Anywhere

This option allows the user to specify the number of lines-per-page on the compiler listing. 2000000 lines-per-page suppresses autopagination.



Item	Description/Default	Range Restrictions
lines per page	integer numeric constant	20 thru MAXINT

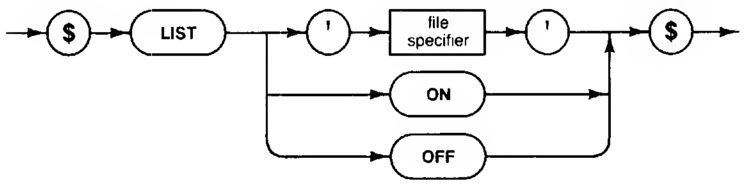
Example

```
$lines 55$
$lines 2000000$    (*suppress autopagination*)
```

LIST

Default: ON to Std. output file
Location: Anywhere

This option controls whether or not a listing is being generated, and where it is being directed to.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier

Semantics

"LIST" is interpreted as "LIST ON".

LIST with a file specifier specifies that the file is to receive the compilation listing.

LIST OFF suppresses listing.

LIST ON resumes listing. No listing will be produced at all, regardless of this option, unless requested by the operator when the Compiler is invoked. (i.e. the "-L" option of the *pc* command is specified.)

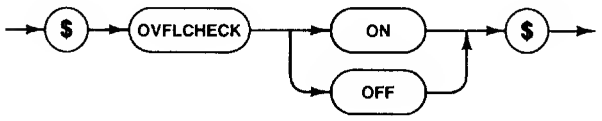
Example

```
$list '/users/steve/keeplist'$
$list off$
```

OVFLCHECK

Default: ON
Location: Statement-by-statement

This option gives the user some control over overflow checks on arithmetic operations.



Semantics

"OVFLCHECK" is interpreted as "OVFLCHECK ON"

ON specifies that overflow checks will be emitted for all in-line arithmetic operations.

OFF does not suppress all checks; they will still be made for 32-bit integer DIV, MOD, and multiplication.

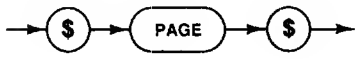
Example

```
$ovflcheck off$
```

PAGE

Default: Not Applicable
Location: Anywhere

This option causes a formfeed to be sent to the listing file if compilation listing is enabled.



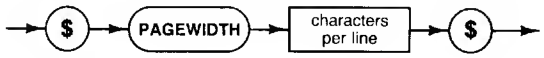
Example

```
$page$
```

PAGEWIDTH

Default: 120
Location: Anywhere

This option allows the user to specify the width of the compilation listing.



Item	Description/Default	Range Restrictions
characters per line	integer numeric constant	80 thru 132

Semantics

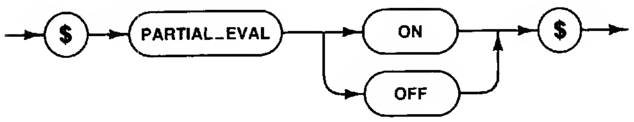
The integer parameter specifies the number of characters in a printer line.

Example

```
$pagewidth 80$
```

PARTIAL_EVAL

Default: OFF
Location: Statement-by-statement



Semantics

"PARTIAL_EVAL" is interpreted as "PARTIAL_EVAL ON".

ON suppresses the evaluation of the right operand of the AND operator when the left operand is FALSE. The right operand will not be evaluated for OR if the left operand is TRUE.

OFF causes all operands in logical operations to be evaluated regardless of the condition of any other operands.

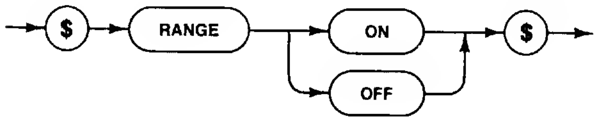
Example

```
$partial_eval on$
while (p<>nil) and (p^.count>0) do
  p := p^.link;
```

RANGE

Default: ON
Location: Statement-by-statement

This options enables and disables run-time-checks for range errors.



Semantics

"RANGE" is interpreted as "RANGE ON".

ON specifies that run time checks will be emitted for array and case indexing, subrange assignment, and pointer dereferencing.

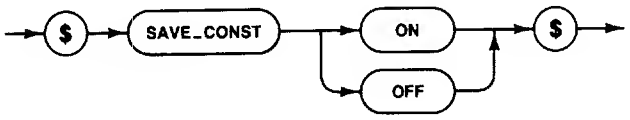
Example

```
var a: array[1..10] of integer; i: integer;
...
i := 11;
$range off$
a[i] := 0;    (* invalid index not caught! *)
```

SAVE_CONST

Default: ON
Location: Anywhere

This option controls whether the name of a structured constant may be used by other structured constants.



Semantics

"SAVE_CONST" is interpreted as "SAVE_CONST ON".

ON specifies that compile-time storage for the value of each structured constant will be retained for the scope of the constant's name (so that other structured constants may use the name).

OFF specifies that storage will be deallocated after code is generated for the structured constant.

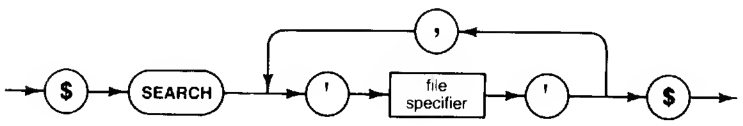
Example

```
$save_const off$
type ary = array [1..100] of integer;
const acon = ary [345,45691, ..... ];
    (*big constants take lots of compile-time memory*)
```

SEARCH

Default: Not Applicable
Location: Anywhere

This option is used to specify files to be used to satisfy IMPORT declarations.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier

Semantics

SEARCH must be the last option in an option list!

Each string specifies a file which may be used to satisfy IMPORT declarations. Files will be searched in the order given. The file, "**/lib/libpc.a**" is always searched last. A default maximum of 9 files may be listed. (See $SEARCH_SIZE ... $.)

Specified files may be either "a.out" or archive (".a") format.

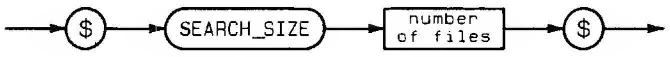
Example

```
$search '/users/steve/firstfile.a','/users/steve/secondfile.a'$
import complexmath, polarmath;
```

SEARCH_SIZE

Default: 9 files
Location: At front

This option allows you to increase the number of external files you may SEARCH during a module's compilation.



Item	Description/Default	Range Restrictions
number of files	integer numeric constant	less than 32767

Semantics

When compiling a Pascal module, it is sometimes desirable to import another module from another file. To import a module from another file, the SEARCH option is used to identify the file. Up to nine SEARCH options may be given unless the SEARCH_SIZE option is given. The SEARCH_SIZE option allows you to SEARCH up to 32 766 external files for imported modules.

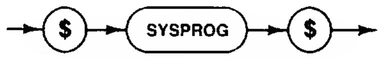
Example

```
$search_size 30$
```

SYSPROG

Default: System Programming Extensions not enabled
Location: At Front

This option makes available some language extensions which are useful in systems programming applications. See "System Programming Language Extensions" in this appendix.



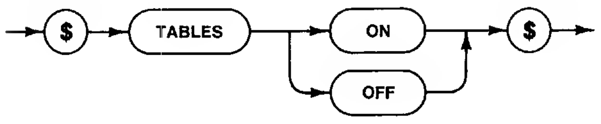
Example

```
$sysprog$
program machinedependent;
...
```

TABLES

Default: OFF
Location: Not in Body

This option turns on and off the listing of symbol tables.



Semantics

"TABLES" is interpreted as "TABLES ON"

ON specifies that symbol table information will be printed following the listing of each procedure. This is useful for very low-level debugging.

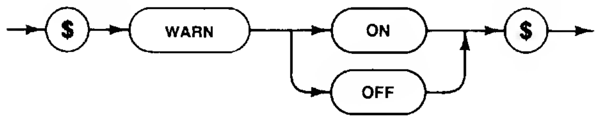
Example

```
$tables$
procedure hasabug (var p: integer);
var
  ...
```

WARN

Default: ON
Location: At Front

This option allows the user to suppress the generation of compiler warning messages.



Semantics

"WARN" is interpreted as "WARN ON" and compiler warnings will be issued.

Example

```
$warn off$
```

Implementation Dependencies

The following list of Pascal keywords have implementation dependencies in Series 200 HP-UX Pascal.

Keyword	Dependency
`append`	You cannot `append` to textfiles. The optional third parameter, the t in `append`(f,s,t), has no signifigence.
`ARRAY .. OF`	There is no limit on the number of elements in an ARRAY.
`close`	The following literals may be used as the optional string parameter in the close procedure. `'LOCK'` or `'SAVE'`: The system will save the file as a permanent file. `'NORMAL'`, `'TEMP'`, or none: If the file is already permanent, it remains in the directory. If the file is temporary, it is removed. `'PURGE'`: The system will remove the file.
Directives	The `external` directive allows Pascal to use externally defined code segments.
`dispose`	See the section on Pascal Heap Managers.
`external`	This directive may be used to indicate a procedure or function that is described externally to the program. See the section: *Pascal and Other Languages.*
Heap Procedures	The supported heap procedures are: `new`, `mark`, `release`, `dispose`. See the Heap Managers Section.
`longreal`	The approximate range is: −1.79769313486231L+308 thru −2.22507385850720L−308, 0, 2.22507385850720L−308 thru 1.79769313486231L+308
`mark`	See the section describing the Pascal Heap Manager
`maxint`	The value of maxint is: 2147483647
`maxpos`	This function always returns `maxint`. (See `lastpos`).
`minint`	The value of minint is: −2147483648
Modules	Module identifiers are restricted to 12 characters.
`real`	The approximate range is: −3.402823E+38 thru −1.175494E-38, 0, 1.175494E−38 thru 3.402823E+38
`release`	Files in the heap will not be closed by release.
`rewrite`	The optional third parameter, the t in `rewrite`(f,s,t), is used for buffered or unbuffered input. See the *Unbuffered Terminal Input* section for details.

Strings	The longest possible string contains 255 characters.
`strread`	The return parameter (indicating the next character to be used with the next `strread` operation) must be an integer (an integer subrange is not allowed).
`strwrite`	The return parameter (indicating the next position to be used with the next `strwrite` operation) must be an integer (an integer subrange is not allowed).
`text`	Appending to a text file is not allowed.
`WITH`	When `f` is a function call, `WITH f DO` is not allowed.

Special Compiler Warnings

The following warnings should never be seen.

```
warning <line number> symbol defined already: <symbol name>
warning <line number> symbol not found: <symbol name>
```

The appearance of these warnings usually indicates a problem with your compiler. The program may not run correctly. If you suspect this to be true, contact your Hewlett-Packard Service Engineer.

Replacements for Pascal Extensions

Over the years, various implementations of Pascal have added extentions to simplify certain operations. One of the more common implementations, the UCSD implementation, added several string functions, byte functions, and IO intrinsics. To simplify the conversion of UCSD Pascal programs to HP Pascal programs for the Series 200 HP-UX operating system, the following table lists replacements for many of the UCSD extensions.

Extension	Replacement
function length	Use strlen and setstrlen
function pos	Use strpos {NOTE: parameters are reversed from pos}
function concat	Use infix "+" operator
function copy	Use str
function delete	Use strdelete
function insert	Use strinsert
function scan	Recode using a FOR loop
function moveleft	Recode using a FOR loop
function moveright	Recode using a FOR loop
function blockread	Recode to use file of buf512 (where: buf512 = PACKED ARRAY[0..511] of char)
function blockwrite	Recode to use file of buf512 (where: buf512 = PACKED ARRAY[0..511] of char)

Other Replacements

Use the following replacements when converting Pascal programs for the Series 200 HP-UX operating system.

`PRINTER:` Use `rewrite(f,'/dev/lp');` Note that use of `/dev/lp` may be restricted by the system. See your system administrator or the *System Administrators Manual* for more information.

`CONSOLE:` Use `input`

`SYSTERM:` Add the following variable declaration:

```
keyboard : text;
```

Then add these procedures to the beginning of the main program:

```
reset(keyboard,'0');
reset(keyboard,'','unbuffered');
```

`IORESULT` Convert to access the variable IORESULT['asm_ioresult']
See the section: *System Programming Language Extensions.*

1 "UCSD Pascal" is a trademark of the Regents of the University of California.

System Programming Language Extensions

Seven extensions to HP Pascal have been provided to support machine-dependent programming and give users better control over (or access to) the hardware.

1. Error Trapping and Simulation
2. Absolute Addressing of Variables
3. Relaxed Typechecking of VAR Parameters
4. The ANYPTR Type
5. Procedure Variables and the Standard Procedure CALL
6. Determining the Absolute Address of a Variable
7. Determining the Size of Variables and Types

These extensions may be used in any compilation which includes the `$SYSPROG ON$` option at the beginning of the text.

The extensions may not be supported by other HP Pascal implementations. The Compiler displays a warning message at the end of compilation when they are enabled.

Error Trapping and Simulation

The TRY-RECOVER statement and the standard function ESCAPECODE have been added to allow programmatic trapping of errors. The standard procedure ESCAPE has been added to allow the generation of soft (simulated) errors.

```
try
  <statement> ;
  <statement> ;
    ...
  <statement>
recover
  <statement>
```

When TRY is executed, certain information about the state of the program is recorded in a marker called the recover-block, which is pushed on the program's stack. The recover-block includes the location of the corresponding RECOVER statement, the height of the program stack, and the location of the previous recover-block if one is active. The address of the recover-block is saved, then the statements following TRY are executed in sequence. If none of them causes an error, the RECOVER is reached, its statement is skipped, and the recover-block is popped off the stack.

But if an error occurs, the stack is restored to the state indicated by the most recent recover-block. Files are closed, and other cleanup takes place during this process. If the TRY was itself nested within another one, or within procedures called while a TRY was active, then the outermost recover-block becomes the active one. Then the statement following RECOVER is executed. Thus, the nesting of TRYs is **dynamic**, according to calling sequence, not statically structured like nonlocal goto's which can only reach labels declared in containing scopes.

The recovery process does not "undo" the computational effects of statements executed between TRY and the error. The error simply aborts the computation, and the program continues with the RECOVER statement.

When an error has been caught, the function ESCAPECODE can be called to get the number of the error. ESCAPECODE has no parameters. It returns an integer error number selected from the error code table.

Escape codes generated by the system are always negative. The programmer can simulate errors by calling the standard procedure `ESCAPE(n)`, which sets the error code to `n` and starts the error sequence. By convention, programmed errors have numbers greater than zero. If an ESCAPE is not caught by a recover-block within the program, it will be reported as an error by the Operating System. Negative values are reported as standard system error messages, and positive values are reported as a halt code value. Note that `HALT(n)` is exactly the same as `ESCAPE(n)`.

TRY-RECOVER statements are usually structured in the following fashion:

```
try
   ....
recover
   if escapecode = (whatever you want to catch)
     then
       begin
          {recovery sequence}
       end
     else
       escape(escapecode);
```

This has the effect of ensuring that errors you **don't** want to handle get passed on out to the next recover-block, and eventually to the system. All programs which are executed are first surrounded by the Operating System with a try-recover sequence. The recovery action for the system is to display an error message.

Absolute Addressing of Variables

A variable may be declared as located at an absolute or symbolically named address. For example,

```
var   ioport [416000]: char;
      assemblysymbol ['asm_external_name']: integer;
```

Each variable named in a declaration may be followed by a bracketed address specifier. An integer constant specifier gives the absolute address of the variable. A quoted string literal gives the name of a load-time symbol which will be taken as the location of the variable; such a symbol must be global in assembly-language which will be loaded with the program.

Absolute addressing with integer constants has little meaning to "virtual memory" operating systems such as HP-UX. However, symbolic addressing can be very useful, as demonstrated in the next section.

Determining IO Errors
When errors are trapped and handled programmatically, by the TRY...RECOVER mechanism, it is often useful to know the exact cause of the error (so that the appropriate response can be taken). Since these errors occur "outside" the program, a method of accessing the error-code from within the program is needed. By adding the following declaration to your program, the last IO error can be accessed.

```
VAR
  IORESULT['asm_ioresult']  :  integer;
```

If you include this declaration within your program, you can test for some errors. For example, suppose you try to reset a file (inside a TRY...RECOVER block). When you check the standard function ESCAPECODE, it returns −10 (indicating an IO error has occurred). You can now check IORESULT and take the appropriate action.

The list of IORESULT values is included at the end of this appendix.

This feature may not be supported on future implementations.

Relaxed Typechecking of VAR Parameters

The ANYVAR parameter specifier in a function or procedure heading relaxes type compatibility checking when the routine is called. This is sometimes useful to allow libraries to act on a general class of objects. For instance an I/O routine may be able to enter or output an array of arbitrary size.

```
type
  buffer = array [0..maxint] of char;
var
  a1: array [2..50] of char;
  a2: array [0..99] of char;

procedure output_hpib(anyvar ary:buffer; lobound,hibound:integer);
   ....

output_hpib(a1,2,50);
output_hpib(a2,0,99);
```

ANYVAR parameters are passed by reference, not by value; that is, the address of the variable is passed. Within the procedure, the variable is treated as being of the type specified in the heading.

This can be very dangerous! For instance, if an array of 10 elements is passed as an ANYVAR parameter which was declared to be an array of 100 elements, an error will very likely occur. The called routine has **no way** to know what you actually passed, except perhaps by means of other parameters as in the example above. ANYVAR should only be used when it's absolutely required, since it defeats the Compiler's normal type safety rules.

Programs calling routines with ANYVAR parameters should be very thoroughly debugged.

The ANYPTR Type

Another way to defeat type checking is with the non-standard type ANYPTR. This is a pointer type which is assignment-compatible with all other pointers, just like the constant NIL. However, variables of type ANYPTR are not bound to a base type, so they can't be dereferenced. They may only be assigned or compared to other pointers. Passing as a value parameter is a form of assignment.

```
type
   p1 = ^integer;
   p2 = ^record
           f1,f2: real;
        end;
 var
   v1,v1a: p1;  v2: p2;
   anyv: anyptr;
   which: (type1,type2);
begin
  new(v1);  new(v2);
  ...
  if ... then
    begin  anyv := v1;  which := type1  end
  else
    begin  anyv := v2;  which := type2  end;
  ...
  if which = type1 then
    begin
      v1a := anyv;
      v1a^ := v1a^ + 1;
    end;
end;
```

This can be very dangerous! The Compiler has no way to know if ANYPTR tricks were used to put a value into a normal pointer. If a pointer type which is bound to a small object has its value tricked into a pointer bound to a large object, subsequent assignment statements which dereference the tricked pointer may destroy the contents of adjacent memory locations.

Programs using this feature must be very thoroughly debugged.

Procedure Variables and the Standard Procedure CALL

Sometimes it is desirable to store in a variable the name of a procedure, and then later to call that procedure.

A variable of this sort is called a procedure variable. The "type" of a procedure variable is a description of the parameter list it requires. That is, a procedure variable is bound to a particular procedure heading.

```
type  procvar = procedure (op:integer);
var   p: procvar;

procedure q(op:integer);    {identically structured parameter list}
  ...

p := q;        {p gets the name of q; in effect p points to q}
call(p,i);     {name of proc variable, then appropriate parameter list}
```

A procedure variable is "called" by the standard procedure CALL, which takes the procedure variable as its first parameter, and a further list of parameters just as they would be passed to a real procedure having the corresponding specification.

It is not possible to create a function variable, that is, a variable which can hold the name of a function.

Don't assign the name of an inner (non-global) procedure to a procedure variable which isn't declared in the same block as the procedure being assigned. Such a variable might be called later, after exiting the scope in which the procedure was declared. The appropriate static link would be missing, yielding unpredictable results.

Determining the Absolute Address of a Variable

The ADDR function returns the address of a variable in memory as a value of type ANYPTR. It accepts, as an optional second parameter, an integer "offset" expression which will be added to the address; this has the effect of pointing "offset" bytes away from where the variable begins in memory.

```
p := addr(variable);
p := addr(variable,offset);
```

ADDR is primarily used for building or scanning data structures whose shapes are defined at run-time rather than by normal Pascal declarations.

The ADDR function is very dangerous! It has the same dangers described above for ANYPTRs, in addition to some of its own. Use of the "offset" can produce a pointer to almost anywhere. This can be dangerous to the integrity of the task's memory.

Never use ADDR to create pointers to the local variables of a procedure or function. When the routine exits, storage for local variables is recovered thus making the value returned by ADDR useless.

Programs using this feature must be very carefully debugged.

Determining the Size of Variables and Types

The size (in bytes) of a type or variable can be determined by the SIZEOF function.

```
n := sizeof(variable);
n := sizeof(typename);
```

If the variable or type is a record with variants, an optional list of tagfield constants may follow the parameter. This is similar to the procedure `new` (although `new` implies that space is to come from the heap).

```
n := sizeof(varrec,true,blue);
```

SIZEOF is not really a function, although it looks like one; it is actually a form of compile-time constant.

Memory Allocation for Pascal Variables

Here is a list of storage allocations for common Pascal data types.

Type	Allocation
`boolean:`	One byte, 0-false 1-true
`character:`	One byte, ASCII character values 0 thru 255
`Enumerated scalar:`	Two bytes, unsigned.
`integer:`	Four bytes signed, −2147483648 to 2147483647
`longreal:`	Eight bytes, approximate range is: ±1.1797693134862315L+308 thru ±2.2250738585072O2L−308
`Pointer:`	Four bytes containing 24-bit logical address.
`Procedure:`	Eight bytes containing address and static nesting information.
`real:`	Four bytes, approximate range is: ±3.40823E+38 thru ±1.175494E−38
`SET:`	Two bytes of length plus multiples of 2 bytes to contain possible elements which require 1 bit each to a maximum of 256 elements.
`String:`	One byte of length field plus up to 255 bytes
`Subrange:`	Two bytes if maximum and minimum values are in [−32768..32767].

Special use of RESET and REWRITE

It is sometimes desirable to create an HP-UX file or pipe from a language other than Pascal, and then call a Pascal routine to continue reading or writing without having to close and then re-open the file. There is a special instance of RESET and REWRITE which make this possible. The first parameter to RESET and REWRITE is the name of the file. The second parameter is the name of an external file. To connect a file or pipe which has been established outside the Pascal program to the file variable, simply put the HP-UX file descriptor in a quoted string as the second parameter. For example:

```
PROGRAM P;
VAR F : TEXT;
BEGIN
RESET(F,'6');
WRITE(F,'ABC');
END.
```

This program will connect the the file variable F with the HP-UX file descriptor 6. The string must contain only the file descriptor; if leading or trailing blanks are present, the string will be interpreted as a file name. No file positioning is done; the file is not rewound. If the file descriptor is associated with a regular file, current position is determined and POSITION(F) is set to this value.

If it is necessary to rewind one of these special files from Pascal, this can be accomplished in either of two ways:

```
PROGRAM P;                    PROGRAM P;
VAR F : TEXT;                 VAR F : TEXT;
BEGIN                         BEGIN
OPEN(F,'6');                  RESET(F,'6');
SEEK(F,1);                    RESET(F);
END.                          END.
```

When attempting to close one of these special HP-UX files, it is not possible to purge it. Even if the "purge" option is specified by CLOSE, the file will be saved.

This feature works for OPEN and APPEND, as well.

Unbuffered Terminal Input

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOF) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information. By default, input from the terminal will behave in this way; that is, it will be buffered into lines.

The HP Pascal Standard requires that input from the standard input device be unbuffered. In order to override the system default of buffered input, the user can add the following statement to his program:

```
REWRITE(INPUT,'','UNBUFFERED');
```

In this mode terminal input is processed in units of bytes. This means that a program attempting to read will receive each byte as it is typed. A line is delimited by a new-line (ASCII LF) character. The end-of-file (ASCII EOF) character behaves the same as if end-of-file was reached while reading from a regular file. To restore the state to buffered input the user can add the following statement to his program:

```
REWRITE(INPUT,'','BUFFERED');
```

The pc Command for Series 200 HP-UX

The *pc* command on the Series 200 HP-UX system is a program (/**bin**/**pc**) that coordinates the execution of the Pascal compiler (/**usr**/**lib**/**pascomp**), the ranlib command, and the linker-loader (/**bin**/**ld**) of the HP-UX system.

When invoked, *pc* parses its arguments. If one of its arguments is a file with a ".p" extension, it proceeds to call the Pascal compiler. The compiler creates an archive, or ".a" file, which contains a ".o" file for each module (See *The Load Format*). It is an archive file, even if there is only one module or main program in the source file. Assuming the compiler was called, the *pc* program then calls the archive utility, ranlib, which causes a directory of ".o" components to be prepended onto the ".a" file.

If the compilation is successful, ranlib is always called, even when the "-c" option is invoked to suppress linking and loading. If ranlib succeeds, *ld* (the link editor) is called, which links the ".a" file with the appropriate library files (/**lib**/**crt0.o**, /**lib**/**libpc.a**, /**lib**/**libc.a**), and any other files which were given as arguments to the *pc* command and are also needed to satisfy unresolved references.

Unless the "−o" option was invoked to cause the final output file to be a particular name, the resulting file is named "a.out", and is ready to run. No matter the pathname of the Pascal source file, the a.out file is left in the current directory from whence *pc* was invoked. If multiple ".p" files are given, the resulting ".a" files will remain in the current directory. If only one ".p" file was given the corresponding ".a" file will be purged, leaving only the a.out file.

See also *pc(1)* in the HP-UX Reference manual.

Using the pc Command

For Series 200 HP-UX, the *pc* command is like the *cc* command. In other words, invoking the Pascal language compiler is very similar to invoking the C language compiler. Notable exceptions are:

- It supports mostly different and fewer options
- It will not accept source files of another language besides Pascal.
- If the -c option is used to suppress linking and loading, a ".a" file is produced, instead of a ".o" file.

The *pc* command can be used to compile Pascal source files, or to link any ".a" or ".o" files that require loading with Pascal run-time support. The *pc* command will accept any combination or number of ".p", ".a", and ".o" files. Usually a compile will go all the way to an "a.out" file, which is linked and loaded.

The Load Format

Here are some things to know about the load format of Pascal programs.

The Series 200 HP-UX HP Standard Pascal compiler (**/usr/lib/pascomp**) produces code that is formatted into archive files. Each module in the source causes a ".o" file to be generated, which is then collected with all ".o" files of a single compilation (a compilation of a single ".p" file), and archived into a ".a" file. Information on archive files and "a.out" format files can be found in the HP-UX manual.

This arrangement permits mixing and matching of object code modules for different Pascal source "modules", using the ar command. The name of each ".o" file is taken from the module name in the source. For purposes of creating this ".o" file, the name can be no longer than twelve characters in length. The compiler treats the main program as a module also. If the name of the program is longer than 12 chars (which is allowed by the compiler), the name is truncated to 12 before being associated with the ".o" file.

All external symbols (module entry points, exported procedures, global data areas, external procedures, aliased names) appear in the link editor symbol table. For user programs, different types of symbols are created by different conventions, and are shown in the following table:

Symbol type	**Construction**
`global data area`	*<module name>*
`exported procedure`	_*<module name>*_*<proc name>*
`module entry points`	_*<module name>*_*<module name>*
`aliased procedure name`	*<aliased name>*
`structured constants`	*<module name>*_*<constant name>*
`aliased variables`	*<aliased name>*
`external procedure (not aliased)`	_*<proc name>*
`main program entry point`	`_main` and _*<programname>*_*<programname>*

Separate Compilation

The $SEARCH ...$ option must be given arguments that are filenames suffixed with ".a" or ".o", which are files that are results of a compilation by this compiler. The $SEARCH ...$ option looks for ".o" files within the ".a" files. If you desire to combine several ".a" files into one (so fewer files have to be searched) you must use the *ar* command to extract the ".o" files, and then recombine them into another ".a" file.

Note

The *ar* command will archive anything you tell it to, even ".a" files. The compiler is not guaranteed to find ".o" files in a ".a" file that is so constructed.

Loading and linking separately compiled ".a" files can be tricky. The loader will not load from an archive file unless entry points defined in it have been previously entered into the link editor symbol table as undefined. This means that in linking several ".a" files derived from Pascal source, the file with the unresolved reference must be given to the loader before the file with the definition.

An example can be seen below. Assume that the following three source files have been compiled separately, with the "-c" option, to give files `FILE1.a`, `FILE2.a`, and `FILE3.a`.

```
{ FILE1.p }
module one;
export procedure printmess1;

implement
  procedure printmess1;
  begin
    writeln('message 1');
    end;

end.

{ FILE2.p }
module two;
export procedure printmess2;

implement
  procedure printmess2;
  begin
    writeln('message 2');
  end;

end.

{ FILE3.p }
$search 'FILE1.a','FILE2.a'$
program test(output);

import one,two;

begin
  printmess1;
  printmess2;
end.
```

Now load them with "`pc FILE[1-3].a`". The following message results:

```
ld: Undefined external -
        two
        one
        _two_two
        _one_one
        _one_printmess1
        _two_printmess2
```

The undefined symbols were generated by *ld* because of `FILE3.a`, which was loaded last. There are four workarounds for this problem, each of which has its uses.

1. The −u option

   The −u option of the loader causes the symbol that is its argument to be entered as undefined, thus forcing the loading of the code that is associated with the symbol. For the above example,
   `pc -u two -u one -u _two_two -u _one_one -u _one_printmess1`
   `-u _two_printmess2 FILE[1-3].a` causes successful linking and loading.

2. Always compile source

   The *pc* command has been designed to enter all module entry points with the −u option, **only for the ".a" files it creates with that invocation**. When it is practical, this is the easiest method. For instance, "`pc FILE[1- 3].p`" works fine. You can verify that it works by using the −v option of *pc* to see the linker run string.

3. Order of linking

   You can link ".a" files in a particular order. "`pc FILE3.a FILE1.a FILE2.a`" would work. Note that the corresponding source files cannot be compiled in that order.

4. Extract .o files

   Using the ar command, all ".o" files may be extracted and then loaded as they are.
   "`pc one.o two.o test.o`" will load successfully.

Program Parameters

It is often desirable to pass the name of one or more files to a Pascal program. This can be accomplished by the use of "program parameters". On Series 200 HP-UX Pascal, these parameters must be of type `file`. The parameters are specified in the program heading in much the same way that `input` and `output` are specified.

For example, this program has one program parameter named READFILE.

```
PROGRAM file_example(input, output, READFILE );
VAR
  readfile : text;
BEGIN
  reset(readfile);
        .
        .
  read(readfile, . . .);
        .
        .
  close(readfile);
END.
```

The name of the physical file to be used by the program parameter is passed by including it as an argument when executing the program. For example,

`a.out` *<file name>*

Where *<file name>* is the name of a physical file.

Multiple file names can be passed by specifying multiple program parameters and providing the names of the files at the time of execution. Each parameter takes one of the specified files.

In the event that no file name is specified for a program parameter, a file will be created. The file name will be the same as the identifier used as the program parameter (the file name will appear in all uppercase letters regardless of the letter case of the identifier).

Program Arguments

A more traditional HP-UX operating system approach to passing arguments to a program is supported by using routines exported from module ARG.

The ARG module exports several functions. The ARGC function returns a count of the number of arguments in the command line. The ARGV function returns a pointer to an array of pointers to the arguments in the command line. The ARGN function returns any particular argument converted to a Pascal string. In addition, a function with similar purpose to ARGN (PAS_PARAMETERS) is provided for compatibility with Series 500 HP-UX Pascal.

The "arguments" module (listed below) may be imported by your program to allow programmatic access to any arguments specified in the command line. Your program does not require a `$SEARCH ...$` directive to access this module, because it is included in **libpc.a**, which is searched automatically.

```
$SYSPROG,RANGE OFF,OVFLCHECK OFF$
module args;

export

  type
    arg_string255 = string[255];
    argtype = packed array[1..maxint] of char;
    argarray = array[0..maxint] of ^argtype;
    argarrayptr = ^argarray;

  function argv: argarrayptr;
  function argc: integer;
  function argn(n: integer): arg_string255;
  function pas_parameters(n: integer; anyvar p: argtype; l: integer): integer;

implement

var
    argc_value['_argc_value'] : integer;
    argv_value['_argv_value'] : argarrayptr;

const
  value_range_error = -8;

function argv: argarrayptr;
  begin
  argv := argv_value;
  end;

function argc: integer;
  begin
  argc := argc_value;
  end;


function argn(n: integer): arg_string255;
  var
    s: arg_string255;
    i: 0..256;
  begin
  if (n >= argc_value) or (n < 0) then
    escape(value_range_error);
  setstrlen(s,255);
  i := 1;
  while argv_value^[n]^[i] <> chr(0) do
    begin
    s[i] := argv_value^[n]^[i];
    i := i + 1;
    end;
  setstrlen(s,i-1);
  argn := s;
  end;
```

```
    function pas_parameters(n: integer; anyvar p: argtype; l: integer): integer;
      var
        i: integer;
      begin
      if (n >= argc_value) or (n < 0) then
        pas_parameters := -1
      else
        begin
        i := 1;
        while (argv_value^[n]^[i] <> chr(0)) and (i <= l) do
          begin
          p[i] := argv_value^[n]^[i];
          i := i + 1;
          end;
        pas_parameters := i-1;
        while i <= l do
          begin
          p[i] := ' ';
          i := i + 1;
          end;
        end;
      end; {pas_parameters}

  end.
```

Programming Example

The following example demonstrates the use of the ARG module.

```
PROGRAM arg_demo(input,output);

VAR
   f: text;
   line: string[255];
   fname: string[80];

IMPORT arg;

BEGIN
   IF argc > 1 THEN
     BEGIN
       fname := argn(1);
       reset(f,fname);
       WHILE NOT eof(f) DO
         BEGIN
           readln(f,line);
           writeln(line);
         END;
     END;
END.
```

When `argc` indicates an argument has been passed, the program assigns the first argument to a filename. The program then resets the file and lists its contents.

You can test the program with the following command line.

```
a.out argdemo.p
```

The contents of the file will be listed to the screen.

Series 200 HP-UX Pascal Heap Managers

The "heap" is the area of memory from which so-called dynamic variables are allocated by the standard procedure NEW. When a process begins, it has available one area of memory for dynamic data. The Pascal heap access routines (NEW, DISPOSE, MARK, and RELEASE) must share this area of memory with any other memory allocation package (MALLOC) called from the same process.

Conceptually the Pascal heap routines NEW, MARK and RELEASE operate in a purely stack-like fashion. When the program finishes with all the variables above a MARK, a RELEASE is called to move the top of the heap (the next available space) back to the value saved by MARK.

MALLOC does not allocate memory in a true stack fashion. Instead, it allocates the first sufficiently large contiguous reach of free space found in a circular search from the last block allocated or freed. It is possible that a memory allocation could be performed from MALLOC after a MARK is done, yet still have an address which is less than the mark pointer. In this situation, RELEASE would not be able to free this memory. The opposite problem arises when a memory allocation performed from MALLOC before a MARK has a pointer value which is greater than the mark pointer. Here RELEASE would free memory which was allocated before the heap was marked, and may destroy valid data.

Pascal now provides two distinct heap managers. The first is simpler, faster, and requires less memory. The second allows RELEASE and MALLOC to be called from the same process.

HEAP1

Version I does not allow RELEASE to be executed after any MALLOC has been done by the process. Memory which has been allocated to the Pascal heap manager can be returned to the Series 200 HP-UX memory manager by RELEASE, and can then be allocated to any other heap manager (i.e. MALLOC).

NEW(P) allocates exactly enough space for a new dynamic variable, and returns the address of the newly-created dynamic variable in P. This space can be allocated from the Pascal free list, or from memory which has never been allocated in this process. The space cannot be allocated from the free lists of other memory allocation packages.

DISPOSE(P) indicates that the space used by the variable P^ is no longer needed, and can therefore be used when dynamic variables are to be created. This space is returned to the Pascal free list, and the pointer P is set to nil.

MARK(P) causes the first free address in the heap to be assigned to P. The next execution of NEW will allocate memory which begins at the address contained in P.

RELEASE(P) can be done only after a MARK(P) has assigned an address to P. This restores the heap to its state at the moment the statement MARK(P) was executed. All dynamic variables created after the MARK statement are effectively destroyed by RELEASE, and the memory space that they used is freed for new dynamic variables.

HEAP2

Version II permits a process to do any combination of allocates and frees by any of the memory managers. This version performs slower for all heap operations (significantly slower to do a RELEASE), and it requires more space. Once memory has been allocated to the Pascal heap manager, this memory can only be reused by Pascal. The memory is not returned to the Series 200 HP-UX memory manager until the process terminates.

NEW(P) allocates an extra twelve bytes for a new dynamic variable. The first four bytes will contain a forward pointer in a linked list of all currently allocated segments. The next four bytes contain a backward pointer to the most recently allocated memory segment. The last four bytes contain the word size of the current segment. This space can be allocated from the Pascal free list, or from memory which has never been allocated in this process. The space cannot be allocated from the free lists of other memory allocation packages. The address of the newly-created dynamic variable is returned in P.

DISPOSE(P) indicates that the space used by the variable P^ is no longer needed, and can therefore be used when dynamic variables are to be created. This space is returned to the Pascal free list along with the extra twelve bytes which were allocated by NEW(P). The pointer P is set to nil.

MARK(P) allocates twelve bytes to put a marker into the list of all currently allocated segments. The first four bytes will contain a forward pointer in this linked list. The next four bytes contain a backward pointer to the most recently allocated memory segment. The last four bytes contain the word size of the current segment. This space can be allocated from the Pascal free list, or from memory which has never been allocated in this process. The space cannot be allocated from the free lists of other memory allocation packages. The address of the newly-created marker is returned in P.

The next execution of NEW will **NOT** allocate memory which begins at the address contained in P.

RELEASE(P) can be done only after a MARK(P) has created a marker in the list of allocated segments and assigned an address to P. This restores the heap to its state at the moment the statement MARK(P) was executed. It begins with the marker and disposes of all segments following it in the list of allocated segments. All dynamic variables created after the MARK statement are effectively destroyed by RELEASE, and the memory space that they used placed in the Pascal free list. RELEASE will only free memory which has been allocated by NEW and MARK; it does not affect memory which was allocated by any other memory allocation package.

Pitfalls

Pascal standards place certain restrictions on heap operations. You may be able to write a program which let you "get away with" ignoring the following restrictions using Version I, whereas Version II will produce unpredictable results.

- The pointer variable passed to RELEASE must have been generated only by a MARK.
- It is not permissible to RELEASE a pointer which was returned by NEW.
- Pointer variables returned by NEW and MARK can be compared only for equality or inequality. The result of comparing these pointers in any other relation is undefined.

Deciding which Heap Manager to Use

If you have a stand-alone Pascal program which does not call any library routines, then you should use Version I. You will have to use Version II if your program calls both MALLOC and RELEASE. You may not be able to tell whether both are called (either may be called from a library routine). In this case, you should try using Version I first. If you ever get

```
ERROR -31:Calls to RELEASE and MALLOC are incompatible.
```

you should then use Version II.

Specifying the Heap Manager

Version I is automatically included with the Pascal run time support, whether you use the *pc* command or compile in another language and link **/lib/libpc.a**. If you decide to use Version II, you must specify this explicitly, by giving a –l option:

```
pc prog.p -l heap2
```

or

```
pc -c prog.p
cc cprog.c prog.a -l heap2 /lib/libpc.a
```

Note

If `heap2` and **/lib/libpc.a** are both specified, `heap2` MUST precede **/lib/libpc.a.**

Pascal and Other Languages

Series 200 HP-UX Pascal can communicate with other languages on the system. Simple data types, like integers, longreals, and characters are the same for Pascal, C, and Fortran. Therefore, these simple types can be passed to routines written in other languages. Strings and other complex data types cannot be passed between languages, unless great care is taken to construct types that each language can understand and the data types are passed by reference.

Calling Other Languages from Pascal

An external declaration is required to call other languages (including Series 200 HP-UX system calls) from Pascal. Like other compilers on this HP-UX system, this compiler prepends an underscore ("_") on most external symbols (see the previous section: *The Load Format*). If the external name is the same as the one you are going to use in Pascal, then no `$alias...$` is required. If you want to use a different name, then you must also use `$alias "_<proc name>"$` in the procedure heading, prepending an underscore for C, FORTRAN, and Pascal names. Since the assembler does not prepend underscores on symbol names, use one in a `$alias...$` option only if it actually appears in the source.

A program containing an external declaration requires an EXTERNAL directive. The EXTERNAL directive is similar in construction to the FORWARD directive.

```
PROCEDURE elsewhere(i: integer; b: boolean); EXTERNAL;

PROCEDURE $alias '_realproc'$ myproc(i; integer); EXTERNAL;
```

Calling Pascal from Other Languages

Calling Pascal from any other languages **requires** that calls to `asm_initproc` and `asm_wrapup` bracket the program containing calls to Pascal routines. These routines are in assembler and the symbol names are: "`_asm_initproc`" and "`_asm_wrapup`" (they are located in **/lib/libpc.a**). The `initproc` procedure has one parameter that is a pointer to an integer. The integer may be zero (echo) or non-zero (no echo). Only one call to each of these routines is required per program. Among other things, they set up the Pascal file system, heap manager, and error recovery. Without them, results may not be as expected.

Pascal Run Time Error Handling

During the execution of a Pascal program, an error may originate from several sources:

- In-line compiled code
- Miscellaneous run time support routines (String, Set, Math, etc.)
- Pascal file system
- HP-UX file system support (system errors)
- Hardware (SIGNALS)

By using the $SYSPROG$ extensions TRY, RECOVER, and ESCAPECODE, almost all of these errors can be trapped for inspection. A *kill* signal cannot be caught.

In the broadest sense, there are two kinds of errors; errors resulting from the execution of in-line code and errors resulting from calls to support routines "outside" the program. The in-line errors include range violation errors, NIL pointer errors, and math overflow errors.

When a program is compiled, the compiler normally emits calls to an error routine which will generate an escapecode upon the detection of an in-line error. These calls can be suppressed by the use of compiler options. See the compiler options: RANGE and OVFLCHECK.

Errors detected during the execution of miscellaneous run time support routines generate escapecodes the same way that in-line compiled code does. The key difference is that errors detected by support routines cannot have the error generation suppressed.

Errors detected by the Pascal file system (IO errors) are generated by a combination of run time support code and in-line compiled code. The file system detects an error and assigns an appropriate IO error number to a global variable. After each call to a file system routine, the compiler also emits code to test the IO error global variable and conditionally generates an escapecode error of −10. You may access this global variable by adding a declaration to your program. See the *System Programming Language Extensions* section.

During normal execution of the Pascal file system, HP-UX file support routines are continuously called to actually perform the desired actions. In most cases, if an error condition is returned to the Pascal file system, its significance is translated into a Pascal file system IO error. There are, however, conditions which arise that are totally unexpected, and in these cases a SYSTEM error is generated (escapecode of −30). The generation of these errors cannot be suppressed.

The final way in which an error can be generated is by an HP-UX signal. All signals that can be intercepted by a user process are converted into appropriate escapecode values.

When emitting code for a main program, the Pascal compiler first emits a call to an initialization routine. When executed, the initialization routine calls the Pascal procedure `catch_signals` (see listing). The `catch_signals` procedure instructs the operating system to transfer control to the `catch_all` procedure whenever a signal occurs. The `catch_all` procedure determines which signal occurred and generates an appropriate escapecode. While the generation of these errors cannot be suppressed, you can set up your own routine to handle any particular signal desired.

Also see the HP-UX documentation for SIGNAL.

What follows is a complete listing of the signal handling module. A listing of all IO, SYSTEM and ESCAPECODE messages that could be generated appears at the end of this appendix.

```
$sysprog$
  module signals;

export
  procedure catch_signals;

  procedure default_signals;

  procedure catch_all( sig_no: integer; typ: integer; ptr: anyptr );

implement

  type
    shortint = -32768..32767;
    sigvals = (dummy,sighup,sigint,sigquit,sigill,sigtrap,sigiot,sigemt,
               sigfpe,sigkill,sigbus,sigsegv,sigsys,sigpipe,sigalarm,
               sigterm,user1,user2,sigchild,sigpwr);

    sig_proc = procedure(sig_no: integer; typ: integer; ptr: anyptr);

  var
    r : record case integer of
          1: (proc : sig_proc);
          2: (address : anyptr;
              static : integer);
        end;
    asm_sig_no['asm_sig_no'] : integer;

  const
    sigdfl = NIL;

  function signal $ALIAS '_signal'$
              (i: integer; p: anyptr): anyptr; external;

  procedure catch_all( sig_no: integer; typ: integer; ptr: anyptr );
    var
      p : anyptr;
    begin
    r.proc := catch_all;
    asm_sig_no := sig_no;
    p := signal(sig_no,r.address);
    case sig_no of
      ord(sighup):   {hangup}
                     escape(-21);

      ord(sigint):   {interrupt -- break key or ^C }
                     escape(-20);

      ord(sigquit):  {quit -- ^|}
                     escape(-21);
    ord(sigill):    {illegal instruction -- not reset to default}
                    case typ of
                      4: escape(-13);{kludge for temp signals}
                      6: escape(-8);          {chk}
                      7: escape(-4);          {trapv}
                      otherwise escape(-21);
                    end;

    ord(sigtrap):   {trace trap -- not reset to default}
                    escape(-21);

    ord(sigiot):    {linea}
                    escape(-21);

    ord(sigemt):    {unimplemented instruction}
                    escape(-21);

    ord(sigfpe):    {floating point exception and divide by zero}
                    if typ = 5 then
                      escape(-5)          {zerodiv}
                    else
                      escape(-21);

    ord(sigkill):   {cannot be caught};

    ord(sigbus):    {bus error}
                    escape(-12);

    ord(sigsegv):   {address violation}
                    escape(-11);

    ord(sigsys):   {bad arg to system call}
                   escape(-21);

    ord(sigpipe): {write on pipe with no one to read}
                  escape(-21);

    ord(sigalarm):{alarm clock went off}
                  escape(-21);

    ord(sigterm): {software termination -- similar to sigkill}
                  escape(-20);

    ord(user1):   {user defined}
                  escape(-21);

    ord(user2):   {user defined}
                  escape(-21);

    ord(sigchild):{child died -- do not catch this signal} ;

    ord(sigpwr):  {power fail -- will never get to user} ;

  end; {case}
end;
```

```
   procedure catch_signals;
     const
       sig_ign = 1;
     var
       i: shortint;
       rec: record case integer of
              1: (ptr: anyptr);
              2: (i  : integer);
            end;
     begin
     r.proc := catch_all;
     for i := ord(sighup) to ord(sigpwr) do
       begin
       if i <> ord(sigchild) then
         begin
         rec.ptr := signal(i,r.address);   {maintain signals that are ignored}
         if rec.i = sig_ign then
           rec.ptr := signal(i,rec.ptr);
         end;
       end;
     end;

   procedure default_signals;
     var
       i: shortint;
       p: anyptr;
     begin
     for i := ord(sighup) to ord(sigpwr) do
       p := signal(i,sigdfl);
     end;

end.
```

Operating System Run Time Error Messages

Errors detected during the execution of a program generate an integer number. An error message is obtained by scannng the appropriate error message file for a line beginning with the same integer value.

There is nothing to prevent you from modifying the error messages. If the error message file cannot be found or if its contents are invalid, subsequent error messages will be displayed as integer values.

When using the TRY..RECOVER construct, the following numbers correspond to the value of `ESCAPECODE`.

These messages are in the file named: **/usr/lib/escerrs**.

−1	Abnormal termination.
−2	Not enough memory.
−3	Reference to NIL pointer.
−4	Integer overflow.
−5	Divide by zero.
−6	Real math overflow.
−7	Real math underflow.
−8	Value range error.
−9	Case value range error.
−10	Non-zero IORESULT –
−11	Segmentation violation.
−12	CPU bus error.
−13	Illegal CPU instruction.
−14	CPU privilege violation.
−15	Bad argument – SIN/COS.
−16	Bad argument – Natural Log.
−17	Bad argument – SQRT.
−18	Bad argument – real/BCD conversion.
−19	Bad argument – BCD/real conversion.
−20	Stopped by user.
−21	Unassigned CPU trap.
−30	System error –
−31	Calls to RELEASE and MALLOC are incompatible.
−32	Heap operations out of sequence.
−33	Illegal variant on dispose.

IO Errors

When `ESCAPECODE` = −10 one of the following errors has occurred. You can determine which error has occurred if you include the following variable declaration in your program.

```
VAR  IORESULT['asm_ioresult'] : integer;
```

The value of IORESULT will match one of the following errors.

These messages are in the file named: **/usr/lib/ioerrs**.

7	Bad file name.
8	No room on volume.
10	File not found.
13	File not open.
14	Bad input format.
24	File not opened for reading.
25	File not opened for writing.
26	File not opened for direct access.
28	String subscript out of range.
29	Bad file close string parameter.
30	Attempt to read past end-of-file mark.
36	File type illegal or does not match request.
39	Undefined operation for file.

System Errors

The following are HP-UX system error messages.

When using the TRY..RECOVER construct, an `ESCAPECODE` = -30 indicates a system error has occurred.

These messages are in the file named: **/usr/lib/syserrs**.

1	Not owner.
2	No such file or directory.
3	No such process.
4	Interrupted system call.
5	I/O error.
6	No such device or address.
7	Arg list too long.
8	Exec format error.
9	Bad file number.
10	No child processes.
11	No more processes.
12	Not enough space.
13	Permission denied.
14	Bad address.
15	Block device required.
16	Mount device busy.
17	File exists.
18	Cross-device link.
19	No such device.
20	Not a directory.
21	Is a directory.
22	Invalid argument.
23	File table overflow.
24	Too many open files.
25	Not a typewriter.
26	Text file busy.
27	File too large.
28	No space left on device.
29	Illegal seek.
30	Read-only file system.
31	Too many links.
32	Broken pipe.
33	Math argument.
34	Result too large.

Pascal Compiler Errors

Errors detected during the compilation of a program generate an integer number. An error message is obtained by scanning the appropriate error message file for a line beginning with the same integer value.

There is nothing to prevent you from modifying the error messages. If the error message file cannot be found or if its contents are invalid, subsequent error messages will be displayed as integer values.

These messages are in the file named: /**usr**/**lib**/**paserrs**.

1	Erroneous declaration of simple type;
2	Expected an identifier ;
4	Expected a right parenthesis ")";
5	Expected a colon ":";
6	Symbol is not valid in this context;
7	Error in parameter list;
8	Expected the keyword OF;
9	Expected a left parenthesis "(";
10	Erroneous type declaration;
11	Expected a left bracket "[";
12	Expected a right bracket "]";
13	Expected the keyword END;
14	Expected a semicolon ";";
15	Expected an integer;
16	Expected an equal sign "=";
17	Expected the keyword BEGIN;
18	Expected a digit following ".";
19	Error in field list of a record declaration;
20	Expected a comma ",";
21	Expected a period ".";
22	Expected a range specification symbol "..";
23	Expected an end of comment delimiter;
24	Expected a dollar sign "$";
50	Error in constant specification;
51	Expected an assignment operator ":=";
52	Expected the keyword THEN;
53	Expected the keyword UNTIL;
54	Expected the keyword DO;
55	Expected the keyword TO or DOWNTO;
56	Variable expected;
58	Erroneous factor in expression;
59	Erroneous symbol following a variable;
98	Illegal character in source text;
99	End of source text reached before end of program;
100	End of program reached before end of source text;
101	Identifier was already declared;
102	Low bound > high bound in range of constants;
103	Identifier is not of the appropriate class;
104	Identifier was not declared;
105	Non-numeric expressions cannot be signed;
106	Expected a numeric constant here;
107	Endpoint values of range must be compatible and ordinal;
108	NIL may not be redeclared;
110	Tagfield type in a variant record is not ordinal;
111	Variant case label is not compatible with tagfield;
113	Array dimension type is not ordinal;
115	Set base type is not ordinal;
117	An unsatisfied forward reference remains;
121	Pass by value parameter cannot be type FILE;
123	Type of function result is missing from declaration;
125	Erroneous type of argument for built-in routine;

Pascal Compiler Errors (continued)

126	Number of arguments different from number of formal parameters;
127	Argument is not compatible with corresponding parameter;
129	Operands in expression are not compatible;
130	Second operand of IN is not a set;
131	Only equality tests (=, <>) allowed on this type;
132	Tests for strict inclusion (<, >) not allowed on sets;
133	Relational comparison not allowed on this type;
134	Operand(s) are not proper type for this operation;
135	Expression does not evaluate to a boolean result;
136	Set elements are not of ordinal type;
137	Set elements are not compatible with set base type;
138	Variable is not an ARRAY structure;
139	Array index is not compatible with declared subscript;
140	Variable is not a RECORD structure;
141	Variable is not a pointer or FILE structure;
143	FOR loop control variable is not of ordinal type;
144	CASE selector is not of ordinal type;
145	Limit values not compatible with loop control variable;
147	Case label is not compatible with selector;
149	Array dimension is not bounded;
150	Illegal to assign value to built-in function identifier;
152	No field of that name in the pertinent record;
154	Illegal argument to match pass by reference parameter;
156	Case label has already been used;
158	Structure is not a variant record;
160	Previous declaration was not forward;
163	Statement label not in range 0..9999;
164	Target of nonlocal GOTO not in outermost compound statement;
165	Statement label has already been used;
166	Statement label was already declared;
167	Statement label was not declared;
168	Undefined statement label;
169	Set base type is not bounded;
171	Parameter list conflicts with forward declaration;
177	Cannot assign value to function outside its body;
181	Function must contain assignment to function result;
182	Set element is not in range of set base type;
183	File has illegal element type;
184	File parameter must be of type TEXT;
185	Undeclared external file or no file parameter;
190	Attempt to use type identifier in its own declaration;
300	Division by zero;
301	Overflow in constant expression;
302	Index expression out of bounds;
303	Value out of range;
304	Element expression out of range;
400	Unable to open list file;
401	File not found;
403	Compiler error;
404	Compiler error;
405	Compiler error;
406	Compiler error;
407	Compiler error;
408	Compiler error;
409	Compiler error;

Pascal Compiler Errors (continued)

660	String constant cannot extend past text line;
661	Integer constant exceeds the range implemented;
662	Nesting level of identifier scopes exceeds maximum (20);
663	Nesting level of declared routines exceeds maximum (15);
665	CASE statement must contain a non-OTHERWISE clause;
667	Routine was already declared forward;
668	Forward routine may not be external;
671	Procedure too long;
672	Structure is too large to be allocated;
673	File component size must be in range 1..32766;
674	Field in record constructor improper or missing;
675	Array element too large;
676	Structured constant has been discarded (cf. $SAVE_CONST);
677	Constant overflow;
678	Allowable string length is 1..255 characters;
679	Range of case labels too large;
680	Real constant has too many digits;
681	Real number not allowed;
682	Error in structured constant;
683	More than 32767 bytes of data;
684	Expression too complex;
685	Variable in READ or WRITE list exceeds 32767 bytes;
686	Field width parameter must be in range 0..255;
687	Cannot IMPORT module name in its EXPORT section;
688	Structured constant not allowed in FORWARD module;
689	Module name may not exceed 12 characters;
600	Directive is not at beginning of the program;
602	Directive not valid in executable code;
604	Too many parameters to $SEARCH;
605	Conditional compilation directives out of order;
606	Feature not in Standard PASCAL flagged by $ANSI ON;
607	Language feature not allowed;
608	$INCLUDE exceeds maximum allowed depth of files;
609	Cannot access this $INCLUDE file;
610	$INCLUDE or IMPORT nesting too deep to IMPORT <module-name>;
611	Error in accessing library file;
612	Language extension not enabled;
613	Imported module does not have interface text;
614	LINENUM must be in the range 0..65535 ;
620	Only first instance of routine may have $ALIAS;
621	$ALIAS not in procedure or function header;
646	Directive not allowed in EXPORT section;
647	Illegal file name;
648	Illegal operand in compiler directive;
649	Unrecognized compiler directive;
651	Reference to a standard routine that is not implemented;
652	Illegal assignment or CALL involving a standard procedure;
653	Routine cannot be followed by CONST,TYPE, VAR, or MODULE;
654	Module declaration may not follow structured constant declaration;
655	Record or array constructor not allowed in executable statement;
657	Loop control variable must be local variable;
658	Sets are restricted to the ordinal range 0 .. 255;
659	Cannot blank pad literal to more than 255 characters;

Pascal Compiler Errors (continued)

696	Array elements are not packed;
697	Array lower bound is too large;
698	File parameter required;
699	32-bit arithmetic overflow;
701	Cannot dereference (^) variable of type anyptr;
702	Cannot make an assignment to this type of variable;
704	Illegal use of module name;
705	Too many concrete modules;
706	Concrete or external instance required;
707	Variable is of type not allowed in variant records;
708	Integer following # is greater than 255;
709	Illegal character in a "sharp" string;
710	Illegal item in EXPORT section;
711	Expected the keyword IMPLEMENT;
712	Expected the keyword RECOVER;
714	Expected the keyword EXPORT;
715	Expected the keyword MODULE;
716	Structured constant has erroneous type;
717	Illegal item in IMPORT section;
718	CALL to other than a procedural variable;
719	Module already implemented (duplicate concrete module);
720	Concrete module not allowed here;
730	Structured constant component incompatible with corresponding type;
731	Array constant has incorrect number of elements;
732	Length specification required;
733	Type identifier required;
750	Error in constant expression;
751	Function result type must be assignable;
900	Error opening code file;
901	Error writing to code file;

Table of Contents

Notes

Implementation Appendix

The Series 200 Workstation

This appendix describes the implementation-specific details of HP Pascal for the Workstation Langauage System on the Series 200 Computers.

The following topics are described in this appendix.

- Compiler Options
- Implementation Dependencies
- Supported Pascal Extensions
- System Programming Language Extensions
- Pascal File Support
- Heap Management
- Error Messages

If you are not already familiar with the Pascal lanuguage, the information presented in this appendix may not be sufficient for you to successfully compile and execute a non-trivial Pascal program. If you have difficulties, please refer to the user manuals and techniques manuals provided with your Series 200 Workstation for more information.

Compiler Options

This section describes the compiler options (compiler directives) you may use with HP Pascal on Series 200 Workstations. When specified, compiler options usually have a default action and restrictions on where they may appear. These restrictions are shown on every page immediately below the option. The explanation of these restrictions is given below.

Restrictions on the Placement of Compiler Directives

Location	**Restriction**
Anywhere:	No restriction.
At front:	Applies to entire source file; must appear before the first "token" in the source file (before PROGRAM, or before MODULE if compiling a list of modules).
Not in body:	Applies to a whole procedure or function; can't appear between BEGIN and END. Good practice to put these options immediately before the word BEGIN, or the procedure heading.
Statement:	Can be applied on a statement-by-statement basis or to a group of statements, by enabling before and disabling after the statements of interest.
Special:	As explained under the particular option.

If a option appears in the interface (import or export) part of a module, it will have effect as the module is compiled. However, the option itself will not become part of the interface specification (export text) in the compiled module's object code and will have no effect in the implement section of the module being compiled.

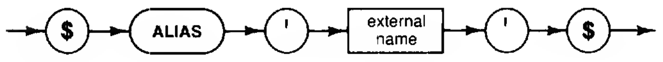
Note

The syntax of the two Compiler options $IF and $SEARCH do not conform to the syntax of all other allowable options.

ALIAS

Default: External name = Procedure Name
Location: Special. See Below

This option causes a name, other than the name used in the Pascal procedure or function declaration, to be used by the loader.



Item	Description/Default	Range Restrictions
external name	string	Entire declaration must fit on one line.

Semantics

The string parameter specifies the external name for the procedure in whose header the option appears.

Example

```
procedure $alias 'charlie'$ p (i: integer);  external;
```

Within the program, calls use the name "p"; but the loader will link to a physical routine called "charlie".

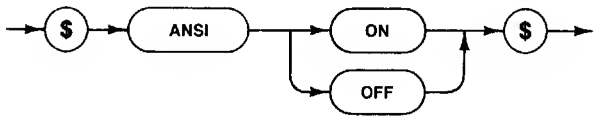
The option must appear between the keywords PROCEDURE or FUNCTION and the first symbol following the semicolon (;) denoting the end of the procedure or function declaration.

The option may not appear in an export section.

ANSI

Default: OFF
Location: At Front

This option selects whether an error message is to be emitted for use of any feature of HP Standard Pascal not contained in ANSI/ISO Standard Pascal.



Semantics

"ANSI" is interpreted as "ANSI ON".

ON causes error messages to be issued for use of any feature of HP Standard Pascal which is not part of ANSI/ISO Standard Pascal.

OFF suppresses the error messages.

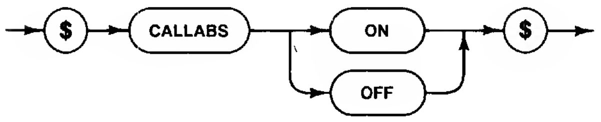
Example

```
$ansi on$
```

CALLABS

Default: ON
Location: Anywhere

This option determines whether 16-bit relative or 32-bit absolute jumps are to be generated by the compiler.



Semantics

"CALLABS" is interpreted as "CALLABS ON".

ON specifies that 32-bit absolute jumps will be emitted for all forward and external procedure calls.

OFF specifies 16-bit PC-relative jumps. Allowed on a statement-by-statement basis.

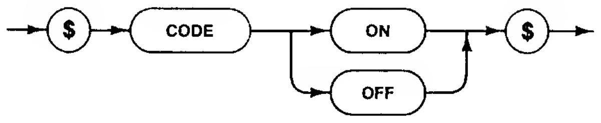
Example

```
$callabs off$
```

CODE

Default: ON
Location: Not in Body

This option is used to control whether a CODE file will be generated by the compiler.



Semantics

"CODE" is interpreted as "CODE ON".

ON specifies that executable code will be emitted.

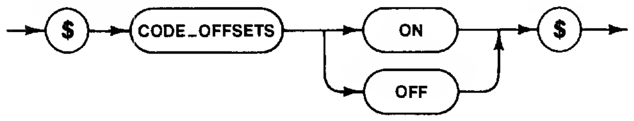
Example

```
$code off$
```

CODE_OFFSETS

Default: OFF
Location: Not in Body

This option controls the inclusion of program counter offsets in the compiler listing.



Semantics

"CODE_OFFSETS" is interpreted as "CODE_OFFSETS ON".

ON specifies that line number-program counter pairs will be printed for each executable statement listed. This can be applied on a procedure-by-procedure basis.

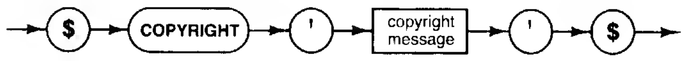
Example

```
$code_offsets on$
```

COPYRIGHT

Default: Not Applicable
Location: Anywhere

This option is provided for inclusion of copyright information.



Item	Description/Default	Range Restrictions
copyright message	string	Entire copyright must fit on one line.

Semantics

The string parameter is placed in the object file as the owner of the copyright. If more than one COPYRIGHT option is included, the last one is effective.

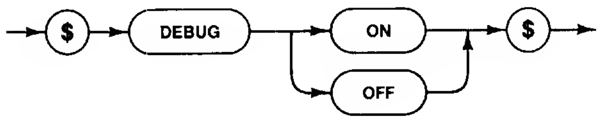
Example

```
$copyright 'Hewlett Packard Company, 1983'$
```

DEBUG

Default: OFF
Location: Not in Body

This option controls whether the code produced by the compiler contains the additional information necessary for the full use of the debugger.



Semantics

"DEBUG" is interpreted as "DEBUG ON"

This option will cause debugging instructions to be emitted by the compiler and may be applied on a procedure-by-procedure basis.

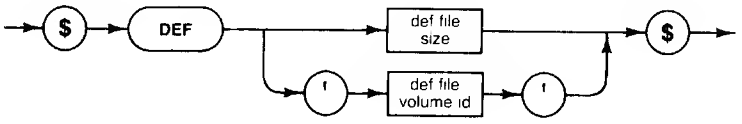
Example

```
procedure buggy;
var  i: integer;
$debug on$
begin
  ...
end;
$debug off$
```

DEF

Default: 10 records (on same volume as code output)
Location: At Front

This option allows the user to change the size and location of the temporary compiler file ".DEF".



Item	Description/Default	Range Restrictions
def file size	integer numeric constant	less than 32767
def file volume id	string	valid volume identifier

If the parameter is a string, it specifies the volume where a temporary Compiler file called ".DEF", which holds external definitions, will be stored. If the parameter is a number, it specifies how many logical records will be allocated for the DEF file. See the section, *What Can Go Wrongi?* near the end of this appendix.

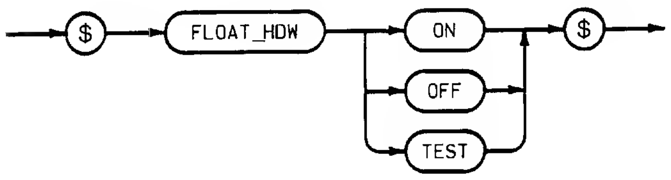
Examples

```
$def 50$
$def 'compvol:'$
$def 'junkvol:', def 50$
```

FLOAT_HDW

Default: OFF
Location: Not in body

This option enables and disables the use of floating-point hardware. **Requires** Pascal 3.0



Semantics

An optional floating-point hardware board is available for Series 200 Computers to increase the execution speed of floating-point math programs.

"FLOAT_HDW" is interpreted as "FLOAT_HDW ON"

ON instructs the compiler to generate accesses to hardware for most floating-point operations. If the hardware does not exist when the program is executed, an error will result.

OFF tells the compiler to generate calls to libraries for all floating-point operations.

TEST causes the compiler to generate both hardware accesses and library calls. The compiler automatically includes code to test for the presence of floating-point hardware. At execution time, if the test succeeds, the hardware accesses are used, otherwise the library calls are used.

The operations that use the hardware include: addition, subtraction, multiplication, division, negation, and the `sqr` function. All other math functions call library routines. There are libraries that access the floating-point hardware. Hardware can also be used by any operation that converts an integer to a real or longreal. The hardware is not used by operations that convert reals or longreals into integers.

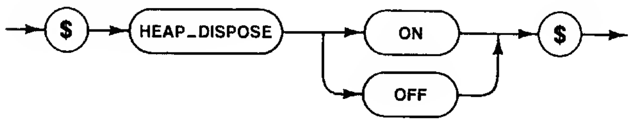
Example

```
$float test$
```

HEAP_DISPOSE

Default: OFF
Location: At Front

This option enables and disables "garbage collection" in the heap.



Semantics

"HEAP_DISPOSE" is interpreted as "HEAP_DISPOSE ON"

ON indicates that DISPOSE allows disposed objects to be reused.

OFF does not recycle disposed objects.

If enabled, this option must appear at the front of the **main program**. It has no effect in separately compiled modules.

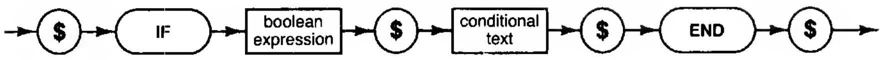
Example

```
$heap_dispose on$
program recycle;
  ...
begin
  dispose(p);  (*free up cell*)
  new(p);      (*probably gets same cell back*)
end.
```

IF

Default: Not Applicable
Location: Anywhere

This option allows conditional compilation.



Item	Description/Default	Range Restrictions
boolean expression	-	may only contain compile time constants
conditional text	source to be conditionally compiled	

Semantics

If the expression evaluates to FALSE, then text following the option is skipped up to the next END option.

If the boolean evaluates to TRUE, then the text following the option is compiled normally.

IF-END option blocks may not be nested.

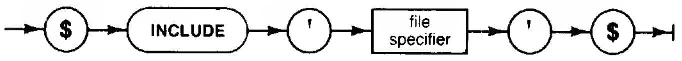
Example

```
const  fancy = true;
       limit = 10;
       size = 9;
...
$if fancy and ((size+1)<limit)$
  ...   (* this will be skipped *)
$end$
```

INCLUDE

Default: Not Applicable
Location: Anywhere

This option allows text from another file to be included in the compilation process.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier

Semantics

The string parameter names a file which contains text to be included at the current position in the program. Included code may contain additional INCLUDE options.

The remainder of the line containing this option must be blank except for the closing "$".

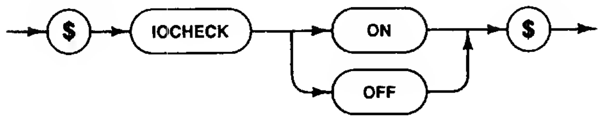
Example

```
PROGRAM inclusive;
 $include 'SOURCE:DECLARS'$
 $include 'SOURCE:BODY'$
END.
```

IOCHECK

Defualt ON
Location: Statement

This option enables and disables error checking following calls to system I/O routines.



Semantics

"IOCHECK" is interpreted as "IOCHECK ON"

ON specifies that error checks will be emitted following calls on system I/O routines such as RESET, REWRITE, READ, WRITE.

OFF specifies that no error will be reported in case of failure.

This option can be used in conjunction with the standard function `IORESULT` if the UCSD or SYSPROG language extensions have been enabled.

IOCHECK can be specified on a statement-by-statement basis.

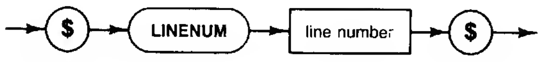
Example

```
$ucsd$
...
$iocheck off$
reset(f,'datafile');
$iocheck on$
if ioresult <> 0 then writeln('IO error');
```

LINENUM

Default: Not Applicable
Location:
Anywhere

This option allows the user to establish an arbitrary line number value.



Item	Description/Default	Range Restrictions
line number	integer numeric constant	1 thru 65535

Semantics

The integer parameter becomes the current line number (for listing purposes and debugging purposes if `$debug$` is enabled).

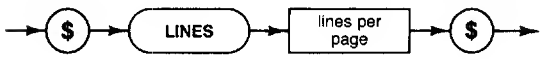
Example

```
$linenum 20000$
```

LINES

Default: 60 lines per page
Location: Anywhere

This option allows the user to specify the number of lines-per-page on the compiler listing.



Item	Description/Default	Range Restrictions
lines per page	integer numeric constant	20 thru MAXINT
string	any valid file specifier	

Semantics

Specifying 2000000 lines-per-page suppresses autopagination.

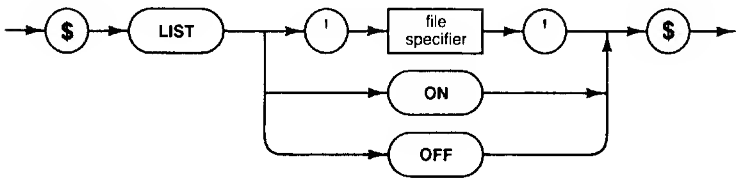
Examples

```
$lines 55$
$lines 2000000$   (*suppress autopagination*)
```

LIST

Default: ON to Std. output file
Location: Anywhere

This option controls whether or not a listing is being generated, and where it is being directed to.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier

Semantics

"LIST" is interpreted as "LIST ON".

LIST with a file specifier specifies that the file is to receive the compilation listing.

LIST OFF suppresses listing.

LIST ON resumes listing. No listing will be produced at all, regardless of this option, unless requested by the operator when the Compiler is invoked. (i.e. the "-L" option of the *pc* command is specified.)

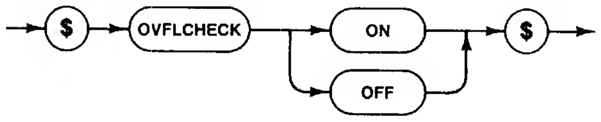
Example

```
$list 'MYVOL:KEEPLIST'$
$list 'PRINTER:'$
$list off$
```

OVFLCHECK

Default: ON
Location: Statement-by-statement

This option gives the user some control over overflow checks on arithmetic operations.



Semantics

"OVFLCHECK" is interpreted as "OVFLCHECK ON"

ON specifies that overflow checks will be emitted for all in-line arithmetic operations.

OFF does not suppress all checks; they will still be made for 32-bit integer DIV, MOD, and multiplication.

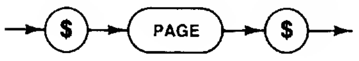
Example

```
$ovflcheck off$
```

PAGE

Default: Not Applicable
Location: Anywhere

This option causes a formfeed to be sent to the listing file if compilation listing is enabled.



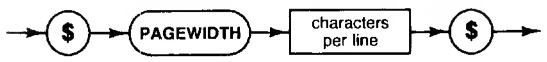
Example

```
$page$
```

PAGEWIDTH

Default: 120
Location: Anywhere

This option allows the user to specify the width of the compilation listing.



Item	Description/Default	Range Restrictions
characters per line	integer numeric constant	80 thru 132

Semantics

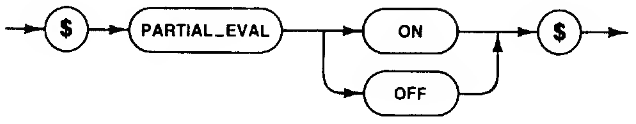
The integer parameter specifies the number of characters in a printer line.

Example

```
$pagewidth 80$
```

PARTIAL_EVAL

Default: OFF
Location: Statement-by-statement



Semantics

"PARTIAL_EVAL" is interpreted as "PARTIAL_EVAL ON".

ON suppresses the evaluation of the right operand of the AND operator when the left operand is FALSE. The right operand will not be evaluated for OR if the left operand is TRUE.

OFF causes all operands in logical operations to be evaluated regardless of the condition of any other operands.

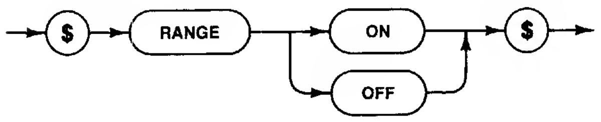
Example

```
$partial_eval on$
while (p<>nil) and (p^.count>0) do
  p := p^.link;
```

RANGE

Default: ON
Location: Statement-by-statement

This options enables and disables run-time-checks for range errors.



Semantics

"RANGE" is interpreted as "RANGE ON".

ON specifies that run-time checks will be emitted for array and case indexing, subrange assignment, and pointer dereferencing.

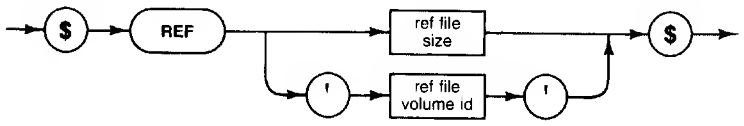
Example

```
var a: array[1..10] of integer; i: integer;
...
i := 11;
$range off$
a[i] := 0;     (* invalid index not caught! *)
```

REF

Default: 30 records (on same volume as code output)
Location: At Front

This option allows you to change the size and location of the temporary compiler file ".REF".



Item	Description/Default	Range Restrictions
ref file size	integer numeric constant	less than 32767
ref file volume id	string	valid volume identifier

If the parameter is a string, it specifies the volume where a temporary Compiler file called ".REF", which holds external references, will be stored. If the parameter is a number, it specifies how many logical records will be allocated for the REF file. See *What Can Go Wrong?* near the end of this appendix.

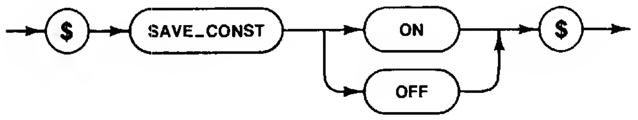
Examples

```
$ref 20$
$ref 'REFVOL:'$
$ref 'JUNKVOL:', ref 50$
```

SAVE_CONST

Default: ON
Location: Anywhere

This option controls whether the name of a structured constant may be used by other structured constants.



Semantics

"SAVE_CONST" is interpreted as "SAVE_CONST ON".

ON specifies that compile-time storage for the value of each structured constant will be retained for the scope of the constant's name (so that other structured constants may use the name).

OFF specifies that storage will be deallocated after code is generated for the structured constant.

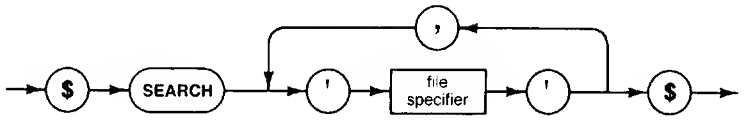
Example

```
$save_const off$
type ary = array [1..100] of integer;
const acon = ary [345,45691, ..... ];
    (*big constants take lots of compile-time memory*)
```

SEARCH

Default: Not Applicable
Location: Anywhere

This option is used to specify files to be used to satisfy IMPORT declarations.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier

Semantics

SEARCH must be the last option in an option list!

Each string specifies a file which may be used to satisfy IMPORT declarations. Files will be searched in the order given. The file, "*LIBRARY" is always searched last. A default maximum of 10 files may be listed. (See $SEARCH_SIZE ... $.)

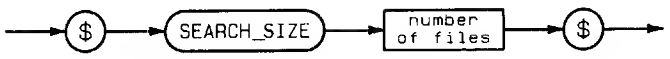
Example

```
$search 'FIRSTFILE','SECONDFILE'$
import complexmath, polarmath;
```

SEARCH_SIZE

Default: 10 files
Location: At front

This option allows you to increase the number of external files you may SEARCH during a module's compilation.



Item	Description/Default	Range Restrictions
number of files	integer numeric constant	less than 32767

Semantics

When compiling a Pascal module, it is sometimes desirable to import another module from another file. To import a module from another file, the SEARCH option is used to identify the file. Up to ten SEARCH options may be given unless the SEARCH_SIZE option is given. The SEARCH_SIZE option allows you to SEARCH up to 32 766 external files for imported modules.

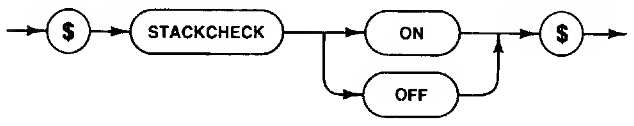
Example

```
$search_size 30$
```

STACKCHECK

Default: ON
Location: Not in Body

This option enables and disables stack overflow checks.



Semantics

"STACKCHECK" is interpreted as "STACKCHECK ON".

ON specifies that stack overflow checks will be generated at procedure entry. It is very dangerous to turn overflow checks off! Obscure and unreported errors may result.

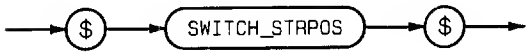
Example

```
$stackcheck off$
procedure unsafe;
var
 may_smash_heap: array [1..500] of integer;
begin ... end;
```

SWITCH_STRPOS

Default: OFF
Location: Anywhere

This option reverses the positions of the parameters of the STRPOS function.



Semantics

Without this option, the order of parameters for the STRPOS function is as follows:

```
STRPOS(search_pattern, source_string)
```

Later, when the HP Pascal Standard was established, the order of parameters was reversed. Thus, if you use the STRPOS function, the compiler issues a harmless warning to indicate that you are not conforming to the standard.

If you wish to conform to the standard, include the `$switch_strpos$` option and reverse the order of the parameters. (See example below.)

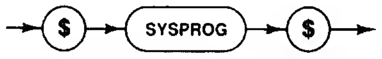
Example

```
$switch_strpos$
...
STRPOS(source_string, search_pattern);
...
STRPOS('i','hurricane');
```

SYSPROG

Default: System Programming Extensions not enabled
Location: At Front

This option makes available some language extensions which are useful in systems programming applications. See "System Programming Language Extensions" in this appendix.



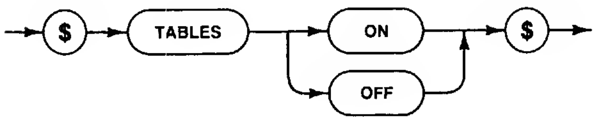
Example

```
$sysprog$
PROGRAM machinedependent;
 ...
```

TABLES

Default: OFF
Location: Not in Body

This option turns on and off the listing of symbol tables.



Semantics

"TABLES" is interpreted as "TABLES ON"

ON specifies that symbol table information will be printed following the listing of each procedure. This is useful for very low-level debugging.

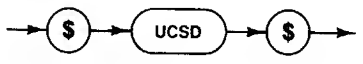
Example

```
$tables$
procedure hasabug (var p: integer);
var
  ...
```

UCSD

Default: UCSD not enabled
Location: At Front

This option allows the compiler to accept most UCSD Pascal language extensions. See *Supported Features of UCSD Pascal* later in this appendix.



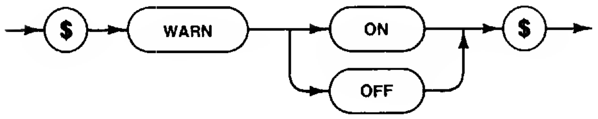
Example

```
$ucsd$
program funnyio;
var
  f: file;   (* no type specified! *)
begin
  unitread(8,ary,80,10);
end.
```

WARN

Default: ON
Location: At Front

This option allows the user to suppress the generation of compiler warning messages. **Requires** Pascal 3.0



Semantics

"WARN" is interpreted as "WARN ON" and compiler warnings will be issued.

Example

```
$warn off$
```

Implementation Restrictions

The following HP Pascal keywords and topics have implementation dependencies when using the Series 200 Workstation compiler.

Keyword	Dependency
`CASE`	CASE statements are implemented using a "jump table". This table is organized as an array of 16-bit values, each an "offset" or distance from the head of the statement to the various cases. The number of entries in the table is the **inclusive** range from the lowest to the highest labels in the statement. If the lowest is labeled "1" and the highest is "15000", there will be 15000 entries! The Compiler displays a warning if it decides a CASE statement is unreasonably large and most of the values in the table are absent or correspond to the same case. If you get such a warning, you should probably recode the statement using IF's or a combination of IF's and a smaller CASE statement. Despite the warning, the Compiler will try to generate the statement as written. If the jump table is very large, it may take a **long** time to write to the output file. You may even think the Compiler has gotten hung up somehow, but the warning message indicates this is not the case.
`close`	The following literals may be used as the optional string parameter in the close procedure. `'LOCK'` or `'SAVE'`: The system will save the file as a permanent file. `'NORMAL'`, `'TEMP'`, or none: If the file is already permanent, it remains in the directory. If the file is temporary, it is removed. `'PURGE'`: The system will remove the file.
Compiler Input	Input to the Compiler is normally prepared by the Editor. Files produced by the Editor are text files, that is, they can be read as files of type TEXT. However, they are more restricted in structure than text files produced by Pascal WRITE statements. Text files are stored as "pages" consisting of 1024 bytes per page. The restriction imposed by the Editor is that no line ever crosses a page boundary; instead, when a line is too long to fit into the current page, the page is padded with Null characters (ASCII zero) and the line which would have spanned the boundary between two pages starts at the front of the next page. WRITE statements simply do not impose this restriction. The Compiler is unable to properly process a line which spans a page boundary. It will "see" spurious characters in the line, and report a syntax error. If you wish to compile a text file not produced by the Editor, the easiest way to fix it is to simply fetch it with the Editor and immediately save it back out. The Editor will fix things up.

Directives — The `external` directive allows Pascal to use externally defined code segments.

`dispose` — See the section on the Pascal Heap Manager.

`external` — This directive may be used to indicate a procedure or function that is described externally to the program. See the section: *Pascal and Other Languages.*

Global Variables — The **global** variables of a program or module must not exceed 65 536 bytes of space.

Global areas are accessed through the processor's A5 register. The A5 register actually points to a location 32 768 bytes below the start of global space. By adding (subtracting) a displacement value (which can range from −32 768 through +32 767) to the contents of register A5, all 65 536 bytes of global space can be accessed.

Use the main command level's `V` command to see the current amount of global space (and free space) available for programs and libraries.

Every module loaded is allocated global area at load time. The sum of global space for all the modules and programs loaded at any time can't exceed 65 536 bytes. About 2000 bytes of global space are taken up by the operating system. The Compiler and Assember each take about 7000 bytes, the Editor about 4000 bytes, the Librarian about 2000 bytes, and the Filer about 1000 bytes.

If you're writing a program which needs a very large global area (i.e. a big array), it can be allocated out of the heap by a call to `new`, then referenced through a pointer. This is a bit of a nuisance, but carries a negligible performance penalty.

```
program bigarray;
type
  gigantic = array [1..20000] of real;  {needs 160,000 bytes }
  ptr = ^gigantic;
var bigthing: ptr;
  i,j: integer;
begin
  new(bigthing);
  for i := 1 to 20000 do bigthing^[i] := 0.0;
end.
```

Note

Each time you permanently load (P-load) a program or library, there will be fewer bytes of global space for use by an application program. The only way to regain the global space is to reboot.

Heap Procedures	The supported heap procedures are: `new`, `mark`, `release`, `dispose`. See the Heap Managers Section.
`IMPORT`	Unless the `$SEARCH_SIZE$` compiler option is specified in your source file, the Compiler can only keep track of a maximum of 10 active input files at once. This means that an INCLUDE file can include another file, and so forth, up to nine times. Exceeding this limit causes errors 608 or 610. When a module is imported which hasn't been previously imported during a compilation, a form of inclusion takes place in which various library files are opened and searched. These files are counted against the maximum of ten while they are open (during the processing of the IMPORT declaration). If module "A" is imported, and its interface specification imports module "B", and so on, the Compiler will chase the importation chain to its very end (unless it runs into the name of a module which has already been seen). If you encounter a situation in which the chain exceeds the limit of ten open input files, you can avoid the problem by making the first module in the chain import all the others in reverse order: the end of the chain first, then the modules which depend on that last one, and so on.
`INCLUDE`	See the restrictions for `IMPORT`.
`integers`	The range is: −2147483648 thru 2147483647
`lastpos`	The `lastpos` function is not implemented on the Workstation.
`linepos`	The standard function `linepos` is not implemented.
Local Variables	The local variables of a procedure or function must not exceed 32 767 bytes of space.
`longreal`	The approximate range is: −1.79769313486231L+308 thru −2.22507385850720L−308, 0, 2.22507385850720L−308 thru 1.79769313486231L+308
`mark`	See the section describing the Pascal Heap Manager
`maxint`	The value of maxint is: 2147483647
`minint`	The value of minint is: −2147483648
Modules	Module identifiers are restricted to 15 characters. No other identifiers are restricted in length in this implementation. **Module Names Used by the Operating System.** If you create a module having the same name as a system module, and your module exports a procedure which has the same name as some procedure exported from that Operating System module, the loader will hook up external references to the wrong place. The simplest way to avoid this is to not use any of the module names in the operating system.

You can use the Librarian to list the file directory of the system modules to discover what names are used by the operating system. In particular, you should check the `INITLIB`, `LIBRARY`, `IO`, `INTERFACE`, and `GRAPHICS` modules.

Some common module names are listed below.

```
CI          MINI        IODECLARATIONS
FS          ASM         KERNEL
KBD         ISR         LOCKMODULE
LOADER      SYSGLOBALS  DEBUGGER
DISCHPIB    UIO         ALLREALS
```

`real`	Type `real` has the same precision as `longreal`. However, in `write` statements the default field width for `longreal` is the same as for `real`, and the exponent is written preceded by `E` instead of `L`. The approximate range is: −1.79769313486231E+308 thru −2.22507385850720E−308, 0, 2.22507385850720E−308 thru 1.79769313486231E+308
`release`	Files in the heap will not be closed by release.
Sets	The ordinal range of sets may not be greater than 256 elements.
Strings	The longest possible string contains 255 characters.
`strread`	The return parameter (indicating the next character to be used with the next `strread` operation) must be an integer (an integer subrange is not allowed).
`strwrite`	The return parameter (indicating the next position to be used with the next `strwrite` operation) must be an integer (an integer subrange is not allowed).
Subrange	A variable declared as a subrange needing 16 or fewer bits for its representation will be stored as a word instead of a longword. For example, `type integer = -32768..32767;` If all the operands of an expression are represented as 16-bit objects, the Compiler implements the expression in 16-bit rather than 32-bit instructions. In particular, integer overflow is detected as a carry into the 17th bit. The rules are: add, subtract: overflow will be detected. divide: −32768 div −1 yields integer overflow. multiply: the result is widened to 32 bits. Note that the representation of an unpacked subrange of integer always reserves room for a sign bit. Hence the range 0..65535 will not be represented in 16 bits, even though it could in fact be.
`text`	Appending to a text file is not allowed.
`WITH`	When `f` is a function call, `WITH f DO` is not allowed.

Pascal Extensions

Over the years, various implementations of Pascal have added extentions to simplify certain operations. One of the more common implementations, the UCSD[1] implementation, added several string functions, byte functions, and IO intrinsics. The Workstation implementation allows you to use the UCSD extensions by including the `$UCSD$` compiler option in your program.

HP Pascal will not provide perfect compatibility with UCSD Pascal or IEM Pascal (HP 9835/9845 systems). In particular, it isn't possible to directly interpret P-code programs since HP Pascal translates programs directly into the native language of the processor. In addition, it is not possible to provide complete compatibility due to definition conflicts between UCSD Pascal and HP Pascal. Most programs should port easily, but some programmer attention will be required.

To simplify the conversion of UCSD Pascal programs to HP Pascal programs for the Series 200 Workstation, the the next section lists many of the UCSD extensions and possible replacements.

1 "UCSD Pascal" is a trademark of the Regents of the University of California.

Supported Features of UCSD Pascal

To use these language extensions, precede the source program text with the $UCSD$ option. HP Pascal replacements for these extensions are given where possible.

`blockread`

This non-standard predefined integer function transfers data from a disc file to an array.

Examples:

```
count := blockread(file_id, array_id, num_blocks);
count := blockread(file_id, array_id, num_blocks, block_num);
count := blockread(file_id, array_id[indx],
                                        num_blocks, block_num);
```

Where `file_id` is the name of an untyped file, `array_id` is the name of an array, and `num_blocks` is the number of 512-byte blocks to be transferred. The optional `block_num` parameter specifies the offset (starting with zero) into the file where the transfer should start. If `block_num` is omitted, the transfer will start at the current position in the file window. The optional `indx` parameter specifies the first element of the array to be accessed by the transfer. The function returns an integer value indicating the actual number of blocks transferred.

Replacement: Recode to use file of buf512 (where: `buf512 = PACKED ARRAY[0..511] of char`).

`blockwrite`

This non-standard predefined integer function transfers data from an array to a disc file.

Examples:

```
count := blockwrite(file_id, array_id, num_blocks);
count := blockwrite(file_id, array_id, num_blocks, block_num);
count := blockwrite(file_id, array_id[indx],
                                        num_blocks, block_num);
```

Where `file_id` is the name of an untyped file, `array_id` is the name of an array, and `num_blocks` is the number of 512-byte blocks to be transferred. The optional `block_num` parameter specifies the offset (starting with zero) into the file where the transfer should start. If `block_num` is omitted, the transfer will start at the current position in the file window. The optional `indx` parameter specifies the first element of the array to be accessed by the transfer. The function returns an integer value indicating the actual number of blocks transferred.

Replacement: Recode to use file of buf512 (where: `buf512 = PACKED ARRAY[0..511] of char`).

`CASE`

In HP Pascal you must add an `OTHERWISE` clause to a `CASE` statement to trap illegal selectors.

In UCSD Pascal, if the selector of a CASE statement doesn't match any of the labelled cases, the entire statement is skipped. HP Pascal instead reports error -9, "Case statement range error".

This problem can be avoided by putting an OTHERWISE clause at the end of the case statement:

```
case i of
  1:  writeln('case 1');
  2:  writeln('case 2');
  otherwise
      writeln('The value of i is ',i:5);
  end;
```

`close`

For HP Pascal, the file options `LOCK`, `NORMAL`, `PURGE`, or `CRUNCH` must be enclosed in quotes.

Comments

UCSD Pascal supports the use of nested comments. This feature can be supported by HP Pascal by using the compiler's `$IF` option.

Comments in UCSD Pascal programs may be delimited by either curly braces or parenthesis-asterisk pairs:

```
{ this is a comment }
(* and so is this *)
```

UCSD Pascal requires that the closing delimiter of a comment be the same "kind" as the opening one. HP Pascal treats the two kinds of opening (and closing) delimiter as synonmyms.

```
(* this is an HP Pascal comment }
(* this is all one { UCSD } comment *)
```

The last example will get a syntax error in HP Pascal because the curly brace after the word "UCSD" terminates the comment.

The easiest way to get around nested comments in a UCSD Pascal program is to surround the outer comment with conditional compilation options:

```
$if false$
  ...   all of the materal inside gets skipped   ...
$end$
```

Compilation Units

The syntax of UCSD Pascal UNITs can readily be changed into an equivalent MODULE for compilation by HP Pascal implementations. The word `INTERFACE` is removed. The word `USES` is replaced by `IMPORT`. And the other declarations in the interface part of the `UNIT` are preceded by the word `EXPORT`.

```
unit goodstuff;                              module goodstuff;
interface                                      import badstuff,betterstuff;
  uses badstuff,betterstuff;                   export
  const                                          const
    ... (constant declarations)                    ...
  type                                           type
    ... (type declarations)                        ...
  var                                            var
    ... (variable declarations)                    ...
  procedure p1 (a,b: integer);                   procedure p1 (a,b: integer);
  function f(x): real;                           function f(x): real;
implementation                               implement
  ...                                          ...
end.                                         end.
```

Compiler Options

The compiler options for UCSD Pascal and HP Pascal differ in syntax. Even if you choose not to convert your UCSD Pascal programs to HP Pascal, you may still need to convert other UCSD compiler options to HP compiler options and include the HP option, `$UCSD$`, at the beginning of your program.

`AUTOPAGE` Use `LINES 2000000` to turn off pagination.

`COPYRIGHT` Supported.

`DEBUG` Supported.

`FLIP` The byteflip option is unsupported (irrelevant).

`GOTO` Unsupported (GOTO's are always allowed).

`IOCHECK` Supported. Also see the `TRY..RECOVER` language extension.

`INCLUDE` Intermixed declarations in INCLUDE are supported.

`LIBRARY` Use the `$SEARCH$` option.

`LINESPERPAGE` Use the `$LINES$` option.

`LINEWIDTH` Use the `$PAGEWIDTH$` option.

`LIST` Use `LIST` *<file specification>* to replace `LIST`, `LIST` *<filename>*, and `LISTFILE`.

`PAGE` Supported.

`QUIET` Unsupported (irrelevant).

`RANGE` Supported.

`SWAP` Unsupported.

`TABLE` Use `$TABLES$`.

`TRACE` Use `$DEBUG$` and use the debugger.

`TRACEPAUSE` Use `$DEBUG$` and use the debugger.

`USERMODE` Unsupported (irrelevant).

`concat`

This non-standard predefined function concatenates any number of strings.

Example: `str_exp := concat(str1, str2, ..., strn);`

Replacement: Use the infix + concatenation operator.

`copy`

This non-standard predefined function returns a string obtained by copying from another string, starting at the specified position.

Example: `str_var := copy(source_str, start_pos, count);`

Where `start_pos` and `count` are integers.

Replacement: Use the `str` function.

`delete`

This non-standard predefined procedure removes a specified number of characters from a string.

Example: `str_var := delete(source_str, start_pos, count);`

Where `start_pos` and `count` are integers.

Replacement: Use the `strdelete` procedure.

`exit`

This non-standard predefined procedure is used to alter program flow.

In UCSD Pascal, the statement `EXIT(proc)` causes normal program flow to be altered. The current procedure is discontinued, and procedures are exited in order (most recently called first) until procedure "proc" is exited. The program continues at the next statement after the call on proc.

This Pascal implementation has no exactly comparable feature; the program must be altered. If the EXIT statement occurs within the procedure which is to be exited, a simple goto statement will suffice. Otherwise you must use the `TRY..RECOVER` statement, which is enabled by the `$SYSPROG$` compiler option.

The basic technique is to surround with a TRY the entire body of any procedure which is the target of an EXIT. The EXIT itself is simulated by calling ESCAPE with an error code corresponding to the name of the procedure to be exited. The target procedure catches this escape in its recovery part and then exits normally.

```
$ucsd$                        $sysprog$
program UCSDexits;            program HPtryrecover;
                              const  exitp2 = 100;  exitp3 =101;
  procedure p1;                 procedure p1;
                                label 1;
  begin                         begin
    ...                           ...
    exit (p1);                    goto 1;  {simple local exit}
    ...                           ...
  end;                        1: end;

  procedure p2;                 procedure p2;
    procedure p3;                 procedure p3;
    begin                         begin
                                    try
      ...                             ...
      exit(p3);                       escape(exitp3);
      ...                             ...
      exit(p2);                       escape(exitp2);
      ...                             ...
                                    recover
                                      if escapecode <> exitp3 then
                                         escape(escapecode);
    end; {p3}                     end; {p3}
  begin {p2}                    begin {p2}
                                  try
    p3;                             p3;
                                  recover
                                    if escapecode <> exitp2 then
                                      escape(escapecode);
  end; {p2}                     end; {p2}

begin {main}                  begin {main}
  p1;                           p1;
  p2;                           p2;
end.                          end.
```

Replacement: This procedure can be simulated by the TRY..RECOVER statement.

`external`

Support: The external directive is supported. Refer to the user manuals for information on using the external directive.

Files

UCSD Pascal doesn't prevent writing to a file which was opened for reading (using RESET). The converse is also true. If you get IO error 24, 25 or 26, the file should have been opened using the HP Pascal standard procedure OPEN.

UCSD Pascal's random access mechanism (SEEK) considers that the first component of a file is number zero. HP Pascal considers that files begin with component number one. The `$UCSD$` option does not fix this problem.

UCSD Pascal recognizes a text file type called `INTERACTIVE`, which differs from files of type `TEXT` in that a component of the file isn't fetched until it is needed. All HP Pascal text files exhibit this "lazy IO" behavior, so you should change INTERACTIVE files to files of type TEXT.

See *Workstation Files* near the end of this appendix for more information on files.

`fillchar`

This non-standard predefined procedure fills a range of memory with a specified value.

Example: `fillchar(variable, count, character);`

Where `variable` may be any type except file, `count` is an integer expression, and `character` is of type `char`.

Replacement: Recode the program using a `FOR` loop. byte stream fill support.

`gotoxy`

This non-standard predefined procedure positions the cursor on the system terminal.

Example: `gotoxy(column,row);`

Replacement: There is no direct replacement for `gotoxy` in HP Pascal. On the Workstation, your program can `IMPORT` the file-system module (`FS`) to access the `fgotoxy` procedure to achieve the same effect.

`halt`

This non-standard procedure terminates the execution of a program.

Example: `halt;`

The halt procedure, with differing syntax, is supported in HP Pascal.

Heap Procedures

See *Heap Management* near the end of this appendix for information on heap procedures.

`insert`

This non-standard predefined procedure inserts a string into another string, at a specified location.

Example: `insert(source_str, dest_str, index);`

Where `source_str` and `dest_str` are string expressions and `index` is an integer.

Replacement: Use the `strinsert` procedure.

`INTERACTIVE`

This file type specifier is disallowed in HP Pascal but the behavior is provided by the TEXT file type.

Integers

HP Pascal integers use 32 bits. You may declare a 16-bit subrange.

Example:

```
TYPE
  int16 : -32768..32767;
```

`ioresult`

This non-standard predefined function returns the result of the last I/O operation. The result value differs for UCSD Pascal and HP Pascal.

`length`

This non-standard predefined function returns the length of a string.

Example: `int_var := length(str_exp);`

Replacement: Use `strlen` and `setstrlen`.

`log`

This non-standard predefined real function returns the decimal logarithm of its parameter.

The `log` function is not supported in HP Pascal.

Replacement: The natural log function, `ln`, is supported. Note that `log(x) = ln(x)/ln(10)`.

Long Integers

Long BCD integers up to 36 digits are not supported by HP Pascal.

`memavail`

This heapspace interrogation function returns the size in bytes, not words.

`moveleft`

This non-standard predefined procedure moves a specified number of bytes, starting with the leftmost byte, to a new location.

Example: `moveleft(source_var, dest_var, count);`

Where source_var and `dest_var` are variables of any type except file. The `count` is an integer expression.

Replacement: Recode the program using a `FOR` loop.

`moveright`

This non-standard predefined procedure moves a specified number of bytes, starting with the rightmost byte, to a new location.

Example: `moveright(source_var, dest_var, count);`

Where source_var and `dest_var` are variables of any type except file. The `count` is an integer expression.

Replacement: Recode the program using a `FOR` loop.

Multiword Comparisons

The multiword comparisons of arrays and records are not supported.

`pos`

This non-standard predefined function returns the position of the first occurrence of a substring within a string.

Example: `int_var := pos(pattern_str_exp, source_str_exp);`

Replacement: Use `strpos`. Note that the parameters are reversed from `strpos`.

Program Heading

A program heading without listing the standard files (i.e `input`, `output`) is supported when the `$UCSD$` option is enabled.

Replacement: Include the standard files in the program heading.

`PWROFTEN`	This non-standard predefined real procedure returns the value of integer powers of ten. This function is not supported. **Replacement:** Use exponentiation.
Reals	This implementation of HP Pascal uses the same internal representation for both `real` and `longreal` types (64-bits). 32-bit reals are not supported.

`scan`	This non-standard predefined function scans a specified section of memory for a specific byte.
	Examples:
	`scan(count, = chr_exp, test_var);`
	`scan(count, <> chr_exp, test_var);`
	Where `count` is the number of bytes to scan, `chr_exp` is an expression which evaluates to a character, and `test_var` is any variable except a file variable. The scan can either match a character (`=`) or not match a character (`<>`).
	Replacement: Recode the program using a `FOR` loop.

`seek`	This non-standard predefined procedure positions the file window in an arbitrary place. **Example:** `seek(file_var, indx);` Where `file_var` is a file variable of a file that was opened using the `open` procedure, and `indx` is the index of the desired component of the file. In HP Pascal the first component's index is one (1), while in UCSD Pascal, the first component's index is zero (0).
`SEGMENT`	UCSD `SEGMENT` procedures are not supported by HP Pascal. Either the entire program must be resident *or* the segmentation procedures supplied with the Series 200 Workstation must be used.
Sets	UCSD Pascal supports sets with up to 4096 elements. HP Pascal sets are limited to 255 elements.

`SIZEOF`	This non-standard predefined integer function returns the number of bytes that a variable uses in memory.
	Examples:
	`num_bytes := sizeof(type_id);`
	`num_bytes := sizeof(var_id);`
	Where `type_id` is a type identifier, and `var_id` is a particular variable.
	Support: This function is supported when system programming language extensions are enabled. (The `$SYSPROG$` compiler option is enabled.)

Standard Units	The standard units: `PRINTER`, `CONSOLE`, and `SYSTERM` are supported. See *Workstation Files* near the end of this appendix for more information.
`str`	This non-standard predefined procedure converts an integer or long integer into a string. **Example:** `str(int_var,str_var);` Where `int_var` is an integer variable, and `str_var` is a string variable. **Replacement:** HP Pascal has the more general procedure `strwrite`. Note: HP Pascal uses this identifier for its "string copy" procedure.

Strings

HP Pascal supports most of the string features available in UCSD Pascal. In UCSD Pascal, the declaration `var s: string` is equivalent to `var s: string[80]`. HP Pascal requires the length specifier.

A similar comment applies to strings value parameters; the specifier `string` is equivalent to the name of an 80-character string type, whereas HP Pascal requires an explicit string typename specifier for value parameters.

UCSD Pascal considers that all strings are compatible as VAR parameters, even if the actual parameter is shorter than the specified formal parameter. This can lead to unexpected bugs. HP Pascal allows two forms of VAR string parameter. If a string typename is used, only another string of identical type may be passed. If the specifier `string` is used, any string may be passed. In the latter case, however, an "invisible" second parameter is also passed, giving the maximum length of the actual parameter. Thus range checking can be performed.

Replacement: In HP Pascal, use the `setstrlen` procedure to set the string length.

Example: `TYPE s = string[maxlength]`

The maximum string length is 255 characters.

```
program UCSDstrings;                 program HPstrings;
type                                 type
  string15 = string[15];               string15 = string[15];
                                       string80 = string[80];
var                                  var
  s1: string;                          s1: string80;
  s2: string [15];                     s2: string15;
  s3: string[80];                      s3: string[80];

  procedure p1 (s: string);            procedure p1 (s: string80);
    ...                                  ...
                                       procedure p1b (s: string);{illegal}
                                         ...
  procedure p2 (s: string15);          procedure p2 (s: string15);
    ...                                  ...
  procedure p3 (var s: string);        procedure p3 (var s: string);
    ...                                  ...
  procedure p4 (var s: string15);      procedure p4 (var s: string15);
    ...                                  ...
                                       procedure p5 (var s: string80);
                                         ...
begin                                begin
  p1(s1);   {legal}                    p1(s1);   {legal}
  p2(s1);   {legal}                    p2(s1);   {legal}
  p3(s1):   {legal}                    p3(s1);   {legal}
  p3(s2);   {legal}                    p3(s2);   {legal}
  p4(s1);   {legal}                    p4(s1);   {illegal}
  p4(s2);   {legal}                    p4(s2);   {legal}
                                       p5(s1);   {legal}
                                       p5(s3);   {illegal}
end.                                 end.
```

`time`

This non-standard procedure or function returns the value of the system's real-time clock.

To read the clock, `IMPORT` the `SYSDEVS` OR `KBD` module and use the `sysclock` procedures and functions.

Type Checking

HP Pascal enforces stricter compatibility rules than UCSD Pascal. HP Pascal generally requires that types be **identical** or **equivalent** where UCSD Pascal will accept mere similarity of form.

```
program UCSDisnotpicky;
type
  complex = record
              re,im: real
            end;
  polar =   record
              r,theta: real
            end;

var
  a: complex;
  b: polar;

begin
  a := b;  { legal }

end.
```

```
program HPispicky;
type
  complex = record
              re,im: real
            end;
  polar =   record
              r,theta: real
            end;
  roundly = polar;
var
  a: complex;
  b: polar;
  c: roundly;
begin
  a := b;  { illegal }
  c := b;  { legal }
end.
```

`UNIT`

A UCSD Pascal `UNIT` is functionally a subset of a HP Pascal `MODULE`. The syntax a little different.

`unitbusy`

This non-standard predefined function tests if an I/O device is busy.

Example: `dev_busy := unitbusy(unit_num);`

Where `unit_num` is an integer expression which evaluates to a valid unit number in the unit-table, and `dev_busy` is a boolean. The function returns true is the device is busy.

`unitclear`

This non-standard predefined procedure resets an I/O device.

Example: `unitclear(unit_num);`

Where `unit_num` is an integer expression which evaluates to a valid unit number in the unit-table.

This operation sets the value of `ioresult`.

`unitread`

This non-standard predefined procedure performs low-level input operations on various devices.

Examples:

```
unitread(unit_num, store_array, count);
unitread(unit_num, store_array, count, block_num);
unitread(unit_num, store_array, count, block_num, async);
unitread(unit_num, store_array[indx], count, block_num, async);
```

Where `unit_num` is the integer identifer of the unit in the unit-table, `store_array` is a packed array in which the data will be stored, and the `count` is the number of bytes to be read.

The optional parameter `block_num` is required for block-structured devices and indicates which block is read. The default is zero. When the optional boolean `async` parameter is true, the transfer is made asynchronously. The default is false.

When specified, the `indx` of the storage array indicates the first element of the array to recieve data.

`unitwait`

This non-standard predefined procedure waits until an I/O operation is finished.

Example: `unitwait(unit_num);`

Where `unit_num` is an integer expression which evaluates to a valid unit number in the unit-table.

`unitwrite`

This non-standard predefined procedure performs low-level output operations on various devices.

Examples:

```
unitwrite(unit_num, store_array, count);
unitwrite(unit_num, store_array, count, block_num);
unitwrite(unit_num, store_array, count, block_num, async);
unitwrite(unit_num, store_array[indx], count, block_num, async);
```

Where `unit_num` is the integer identifer of the unit in the unit-table, `store_array` is a packed array containing the available data, and the `count` is the number of bytes to be written.

The optional parameter `block_num` is required for block-structured devices and indicates which block is written. The default is zero. When the optional boolean `async` parameter is true, the transfer is made asynchronously. The default is false.

When specified, the `indx` of the storage array indicates the first element of the array in which data is available.

Untyped Files

Untyped files are supported with the `$UCSD$` option. Untyped files do not have an associated buffer variable.

Example: `var  un_file : file;`

System Programming Language Extensions

Eight extensions to HP Pascal have been provided to support machine-dependent programming and give users better control over (or access to) the hardware.

1. Error Trapping and Simulation
2. Absolute Addressing of Variables
3. Relaxed Typechecking of VAR Parameters
4. The ANYPTR Type
5. Procedure Variables and the Standard Procedure CALL
6. Determining the Absolute Address of a Variable
7. Determining the Size of Variables and Types
8. The IORESULT Function

These extensions may be used in any compilation which includes the `$SYSPROG ON$` option at the beginning of the text.

The extensions may not be supported by other HP Pascal implementations. The Compiler displays a warning message at the end of compilation when they are enabled.

Error Trapping and Simulation

The TRY-RECOVER statement and the standard function ESCAPECODE have been added to allow programmatic trapping of errors. The standard procedure ESCAPE has been added to allow the generation of soft (simulated) errors.

```
try
  <statement> ;
  <statement> ;
    ...
  <statement>
recover
  <statement>
```

When TRY is executed, certain information about the state of the program is recorded in a marker called the recover-block, which is pushed on the program's stack. The recover-block includes the location of the corresponding RECOVER statement, the height of the program stack, and the location of the previous recover-block if one is active. The address of the recover-block is saved, then the statements following TRY are executed in sequence. If none of them causes an error, the RECOVER is reached, its statement is skipped, and the recover-block is popped off the stack.

But if an error occurs, the stack is restored to the state indicated by the most recent recover-block. Files are closed, and other cleanup takes place during this process. If the TRY was itself nested within another one, or within procedures called while a TRY was active, that previous recover-block becomes the active one. Then the statement following RECOVER is executed. Thus the nesting of TRYs is **dynamic**, according to calling sequence, not statically structured like nonlocal goto's which can only reach labels declared in containing scopes.

The recovery process does not "undo" the computational effects of statements executed between TRY and the error. The error simply aborts the computation, and the program continues with the RECOVER statement.

When an error has been caught, the function ESCAPECODE can be called to get the number of the error. ESCAPECODE has no parameters. It returns an integer error number selected from the error code table. System error numbers are always negative.

The programmer can simulate errors by calling the standard procedure ESCAPE(n), which sets the error code to n and starts the error sequence. By convention, programmed errors have numbers greater than zero. If an ESCAPE is not caught by a recover-block within the program, it will be reported as an error by the Operating System. Negative values are reported as standard system error messages, and positive values are reported as a halt code value. Note that HALT(n) is exactly the same as ESCAPE(n).

TRY-RECOVER statements are usually structured in the following "canonical" fashion:

```
try
   ....
recover
   if escapecode = (whatever you want to catch)
     then
       begin
          {recovery sequence}
       end
     else
       escape(escapecode);
```

This has the effect of ensuring that errors you **don't** want to handle get passed on out to the next recover-block, and eventually to the system. All programs which are executed are first surrounded by the Command interpreter with a try-recover sequence. The recovery action for the system is to display an error message.

Absolute Addressing of Variables

A variable may be declared as located at an absolute or symbolically named address:

```
var   ioport [416000]: char;
      assemblysymbol ['asm_external_name']: integer;
```

Each variable named in a declaration may be followed by a bracketed address specifier. An integer constant specifier gives the absolute address of the variable; this is useful for addressing IO interface hardware. A quoted string literal gives the name of a load-time symbol which will be taken as the location of the variable; such a symbol must be defined (DEF'ed) by an assembly-language module which will be loaded with the program.

Relaxed Typechecking of VAR Parameters

The ANYVAR parameter specifier in a function or procedure heading relaxes type compatibility checking when the routine is called. This is sometimes useful to allow libraries to act on a general class of objects. For instance an I/O routine may be able to enter or output an array of arbitrary size.

```
type
  buffer = array [0..maxint] of char;
var
  a1: array [2..50] of char;
  a2: array [0..99] of char;

procedure output_hpib(anyvar ary:buffer; lobound,hibound:integer);
   ....

output_hpib(a1,2,50);
output_hpib(a2,0,99);
```

ANYVAR parameters are passed by reference. not by value: that is. the address of the variable is passed. Within the procedure. the variable is treated as being of the type specified in the heading.

This can be very dangerous! For instance. if an array of 10 elements is passed as an ANYVAR paramter which was declared to be an array of 100 elements. an error will very likely occur. The called routine has **no way** to know what you actually passed. except perhaps by means of other parameters as in the example above. ANYVAR should only be used when it's absolutely required, since it defeats the Compiler's normal type safety rules.

Programs calling routines with ANYVAR parameters should be very thoroughly debugged. Careless use of this feature can crash your system.

The ANYPTR Type

Another way to defeat type checking is with the non-standard type ANYPTR. This is a pointer type which is assigment-compatible with all other pointers. just like the constant NIL. However. variables of type ANYPTR are not bound to a base type. so they can't be dereferenced. They may only be assigned or compared to other pointers. Passing as a value parameter is a form of assignment.

```
type
   p1 = ^integer;
   p2 = ^record
          f1,f2: real;
         end;
var
   v1,v1a: p1;  v2: p2;
   anyv: anyptr;
   which: (type1,type2);
begin
  new(v1);  new(v2);
  ...
  if ... then
    begin  anyv := v1;  which := type1  end
  else
    begin  anyv := v2;  which := type2  end;
  ...
  if which = type1 then
    begin
      v1a := anyv;
      v1a^ := v1a^ + 1;
    end;
 end;
```

This can be very dangerous! The Compiler has no way to know if ANYPTR tricks were used to put a value into a normal pointer. If a pointer type which is bound to a small object has its value tricked into a pointer bound to a large object, subsequent assignment statements which dereference the tricked pointer may destroy the contents of adjacent memory locations.

Careless use of ANYPTR can crash your system. Programs using this feature must be very thoroughly debugged.

Determining the Absolute Address of a Variable

```
p := addr(variable);
p := addr(variable,offset);
```

The ADDR function returns the address of a variable in memory as a value of type ANYPTR. It accepts, as an optional second parameter, an integer "offset" expression which will be added to the address; this has the effect of pointing "offset" bytes away from where the variable begins in memory. ADDR is primarily used for building or scanning data structures whose shapes are defined at run-time rather than by normal Pascal declarations.

The ADDR function is very dangerous! It has the same dangers described above for ANYPTRs, in addition to some of its own. Use of the "offset" can produce a pointer to almost anywhere, with concommitant dangers to the integrity of system memory.

Never use ADDR to create pointers to the local variables of a procedure or function. Storage for local variables is recovered when the routine exits, so the value returned by ADDR is ephemeral.

Careless use of the pointers returned by ADDR can crash your system. Programs using this feature must be very carefully debugged.

Procedure Variables and the Standard Procedure CALL

Sometimes it is desirable to store in a variable the name of a procedure, and then later to call that procedure. For instance, the system "Unittable" is an array which contains the name of the driver to be called to perform IO on each logical volume.

A variable of this sort is called a "procedure variable". The "type" of a procedure variable is a description of the parameter list it requires. That is, a procedure variable is bound to a particular procedure heading.

```
type  procvar = procedure (op:integer);
var   p: procvar;

procedure q(op:integer);     {identically structured parameter list}
  ...

p := q;            {p gets the name of q; in effect p points to q}
call(p,i);         {name of proc variable, then appropriate parameter list}
```

A procedure variable is "called" by the standard procedure CALL, which takes the procedure variable as its first parameter, and a further list of parameters just as they would be passed to a real procedure having the corresponding specification.

It is not possible to create a "function variable", that is, a variable which can hold the name of a function.

Don't assign the name of an inner (non-global) procedure to a procedure variable which isn't declared in the same block as the procedure being assigned. Such a variable might be called later, after exiting the scope in which the procedure was declared. The appropriate static link would be missing, yielding unpredictable results. See "How Pascal Programs Use the Stack", at the end of this chapter, for an explanation of static links.

Determining the Size of Variables and Types

The size (in bytes) of a type or variable can be determined by the SIZEOF function. This also is enabled by the $UCSD$ option.

```
    n := sizeof(variable);
    n := sizeof(typename);
```

If the variable or type is a record with variants, an optional list of tagfield constants may follow the parameter. This works like the standard Pascal procedure NEW:

```
    n := sizeof(varrec,true,blue);
```

SIZEOF is not really a function, although it looks like one: it is actually a form of compile-time constant.

Memory Allocation for Pascal Variables

Here is a list of storage allocations for common Pascal data types.

TYPE	Allocation
`boolean:`	One byte, 0-false 1-true
`character:`	One byte, ASCII character values 0 thru 255
`Enumerated scalar:`	Two bytes, unsigned.
`integer:`	Four bytes signed, -2147483648 to 2147483647
`longreal:`	Eight bytes, approximate range is: ±1.7976931348623215L+308 thru ±2.225073858507202L-308
`Pointer:`	Four bytes containing 24-bit logical address.
`Procedure:`	Eight bytes containing address and static nesting information.
`real:`	Same as longreal.
`SET:`	Two bytes of length plus multiples of 2 bytes to contain possible elements which require 1 bit each to a maximum of 256 elements.
`String:`	One byte of length field plus up to 255 bytes
`Subrange:`	Two bytes if maximum and minimum values are in [.02332768..32767].

The IORESULT Function

Normally the Compiler emits instructions after each IO statement to verify that the transaction completed properly. If it fails, the program is terminated with an error report.

It is possible to trap IO errors programmatically, using the TRY-RECOVER statement. The system function IORESULT can then be called to discover what went wrong with the transaction.

IO Checks and Results

Normally the Compiler emits instructions after each IO transaction to verify that the transaction completed properly. If it didn't, the program is terminated with an error report. The error code for all IO errors is -10.

You may wish to intercept IO errors programmatically rather than have them terminate the program. This can be done two different ways. The program or module must be compiled with the $SYSPROG$ or $UCSD$ Compiler option at the front of the source text. These options both make available a system function called IORESULT which returns an integer value reporting on the success of the most recent IO transaction. A result of zero indicates a successful transaction; other values are given in the list below.

Method 1. This method is the preferred one. Compile the program or module with $SYSPROG$ enabled, and use the TRY-RECOVER statement to trap the errors.

```
$sysprog$
program trapmethod (input,output);
var
  name: string[80];
  f: text;
  ior: integer;
begin
  repeat
    write('Open what file ? ');
    readln(s);
    try
      reset(f,s+'.text');
      ior := 0;    (*if we get here, it didn't fail*)
    recover
      if escapecode = -10 then  (*it's an IO error*)
        begin
          ior := ioresult;  (*save it; will be affected by write stmt*)
          writeln('  Can''t open it.  IOresult =',ior);
        end
      else
        escape(escapecode);
  until ior = 0;
end.
```

Method 2. This method is used in UCSD Pascal programs. For it to work, you must also suppress the error checks normally emitted by the Compiler.

```
$ucsd$
program ucsdmethod (input,output);
var
  name: string[80];
  f: text;
  ior: integer;
begin
  repeat
    write('Open what file ? ');
    readln(s);
    $iocheck off$
    reset(f,s+'.text');
    $iocheck on$
    ior := ioresult;     (*save it; will be affected by write stmt*)
    if ior <> 0 then
    writeln('  Can''t open it.  IOresult =',ior);
  until ior = 0;
end.
```

The values returned by the `IORESULT` function are listed in the *Error Messages* section at the end of this appendix.

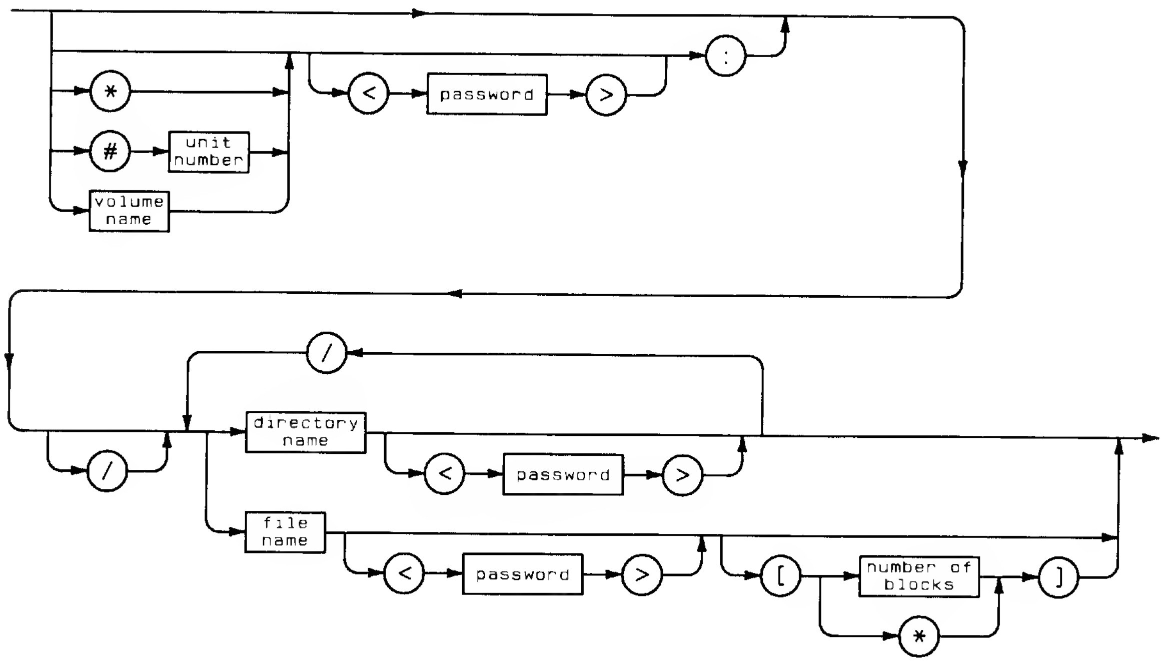
Workstation Files

The file system is covered in detail in the section describing the Filer (file manager) in the user's manual. The abbreviated discussion provided here explains how the connection is made between physical files and Pascal file variables.

A physical file is identified by a **file specification**, which tells what volume the file is on, and further gives the name of the file if the volume is one with a directory. A logical file is simply a file-structured variable declared in a Pascal program. A file variable is associated with a particular physical file when the file is opened by a call to one of the standard procedures RESET, REWRITE or OPEN.

Syntax of Physical File Names

A file specification is a string literal or expression which conforms to the following syntax:



Item	Description/Default	Range Restrictions
unit number	integer; corresponding to an entry in the unit table	1 thru 50
volume name	literal	any legal volume name
password	literal	any legal password
directory name	literal	any legal SRM directory name
file name	literal	any legal file name
number of blocks	integer	any legal number of blocks

The file specifier is a name, one to nine characters long (ten characters if there is no suffix). If you are using a Shared Resource Management (SRM) file system, the file specifier is one to sixteen charcters long including the suffix. See the list of allowable characters below. If the volume specified is an unblocked volume (like PRINTER), which has no directory, the file specifier is ignored.

The file name may end in one of three reserved suffixes:

`.TEXT`	denotes a Pascal text file; usually created by the Editor.
`.CODE`	denotes an executable code file.
`.BAD`	denotes a file spanning a failed region of the mass storage medium.

A file whose name doesn't end in one of these suffixes is generically called a "data" file.

The **size** is used when creating a new file. If it is omitted, the file is created in the largest unused area on the volume. The asterisk syntax allocates either the second largest free area, or half of the largest, whichever is bigger. If a specific size is given, the integer indicates how many 512-byte blocks will be allocated to the file. The size must be at least two blocks, and can't be bigger than the largest free area in the volume. No volume can exceed 32767 blocks, so no file may be bigger than 16,776,704 bytes.

Characters Allowed in Volume and File Names

When specifying file names, letter case is important! The file named `info` is not the same as the file named `INFO`. Also, a file named `stuff.text` will be saved as `stuff.TEXT`, that is, the suffix will be converted by the file system to its uppercase equivalent.

Note

Only the HP Pascal 1.0 Workstation converted all lowercase alphabetic characters to uppercase.

All characters are allowed in names except: control characters (those with ordinal value less than 32), blank " ", sharp "#", asterisk "*", comma ",", colon ":", equals "=", question mark "?", left bracket "[", right bracket "]", and `del` (ordinal 127).

Examples of File Specifications

These examples assume the following variable specifications.

```
var  t: text;
     c: file of char;
     f: file of integer;
```

`reset(t,'MYTEXT.TEXT');`	The `.TEXT` suffix *must* be specified, even though `t` is declared as a textfile. The suffix is part of the name! The file is on the default volume.
`reset(c,'MYTEXT');`	This is a data file on the default volume.
`reset(c,':MYTEXT');`	Same as previous one. An empty volume name is assumed to precede the colon.
`reset(t,'*JUNK.TEXT');`	The file is on the system volume. The colon is optional for a * volume specifier.
`reset(t,'MYVOL:MINE.TEXT');`	The file is on the volume labelled `MYVOL`, wherever that might be found.
`reset(t,'#8:MINE.TEXT');`	The file is on whatever volume is presently in unit #8.
`reset(t,'SYSTERM:');`	Open the keyboard for input.
`rewrite(t,'PRINTER:');`	Open the unblocked volume PRINTER for output.
`rewrite(t,'CONSOLE:');`	Open the CRT volume for output.
`rewrite(t,'#6:');`	Open logical unit #6 for output. The system printer is #6 by convention.
`rewrite(t,'#6');`	The colon is optional.
`rewrite(f,'*JUNK');`	Open a data file called JUNK on the system volume. Allocate the largest free area to this file.
`rewrite(t,'MINE.TEXT[*]');`	Open a text file on the default volume; allocate it half the largest free area.
`rewrite(f,'JUNK[50]');`	Open a data file of 50 blocks.
`open(f);`	The file is opened for both reading and writing. The system will generate a dummy name for it.

Disposition of Files Upon Closing

When a file is closed, its disposition depends on the second parameter to the CLOSE standard procedure.

`close(f,'SAVE')`	The file is made permanent in the volume directory.
`close(f,'LOCK')`	Same as `SAVE`.
`close(f,'NORMAL')`	If the file is already permanent, it remains in the directory. Otherwise it is removed.
`close(f)`	Same as 'NORMAL'.
`close(f,'PURGE');`	If the file was permanent, it is removed from the directory.

Standard Files and the Program Heading

There are four standard files which, if used by your program, are automatically opened when the program starts. Any of these files which is used must be listed in the program heading. No other files should be listed in the heading.

All the standard files are text files.

`INPUT`	The default file for read statements is the keyboard. Characters are echoed to the CRT at the current cursor position as they are read.
`KEYBOARD`	This file also reads from the keyboard, but characters are **not** echoed as they are read. The keystrokes are read straight through, and editing is not enabled.
`OUTPUT`	The default file for write statements. Characters are written to the CRT.
`LISTING`	The default printer file: automatically opened to volume 'PRINTER:' .

Note

The files `INPUT` and `OUTPUT` must *not* be redeclared in the program, while the files `KEYBOARD` and `LISTING` *must* be declared as type `TEXT`. Do not explicitly `close`, `reset` or `rewrite` any of these system files. If they are ever closed, the Initialize command on the main command interpreter prompt will re-open them.

Standard Files Example

```
program use_them_all (input,output,keyboard,listing);
var
   s:  string[80];
  lp:  text;
begin
   rewrite(lp,'PRINTER:');
   readln(s);                     (* from keyboard; echoes to CRT *)
   writeln(s);                    (* to the CRT *)
   readln(KEYBOARD,s);            (* not echoed *)
   writeln(LISTING,s);            (* goes to the printer *)
   writeln(lp,s);                 (* so does this *)
end.
```

File System Differences

To allow for the fact that different computers provide different underlying operating system support, HP Pascal allows certain variations in the parameters passed to the standard procedures for opening and closing files. These parameters appear as strings passed to the standard procedures; it is their content which may vary. For instance, the file naming conventions are very different in different operating systems. Such variations may require minor changes in a program if it is moved to a type of computer different from the one on which it was developed.

When a file is open, its *behavior* in performing the input and output operations of HP Pascal should be the same in all implementations.

Heap Management

The "heap" is the area of memory from which so-called dynamic variables are allocated by the standard procedure NEW. When a program begins running, it has available one area of memory for data. The program's stack begins at the high-address end of this area and grows downward; the heap begins at the low-address end and grows upward. If the stack and heap collide, a Stack Overflow error (escapecode -2) is reported.

Two regimes are available for the recovery of heap variables after they become unwanted: the MARK/RELEASE method, and the DISPOSE method. The first is simpler and faster, the second more general.

MARK and RELEASE

This method uses two standard procedures to manage the heap in a purely stack-like fashion. MARK is called to set a pointer to the next available byte at the top of the heap. Subsequent calls to NEW will all take space from above this point. When the program finishes with all the variables above the mark, RELEASE is called to move the top of the heap (the next available space) back to the value saved by MARK.

```
program markrelease;
type
  ptr =  ^rec;
  rec = record
          f1,f2: integer;
        end;
var
  top,p: ptr;
  i: integer;
begin
  mark(top);          (* remember the base of the heap *)
  repeat
    for i := 1 to 5000 do
      begin
        new(p);       (* allocate from next highest heap address *)
        ...
      end;
    release(top);     (* cut back the heap; recover all space *)
  until false;        (* program will run forever *)
end.
```

When using this method, the computer does not prevent you from making the mistake of releasing to a point **above** the current top-of-heap!

NEW and DISPOSE

Alternatively, the standard procedure DISPOSE can be used to return each unwanted dynamic variable back to a pool of free space.

Calls to DISPOSE will have no effect (the freed storage will not be reused) unless the **main program and the modules containing the NEW and DISPOSE calls** are compiled with the option $HEAP_DISPOSE ON$.

```
program disposal;
type
  ptr = ^ rec;
  rec = record
          next: ptr;
          f1,f2: integer;
        end;
```

```
var
  top,p,root: ptr;
  i: integer;
begin
  mark(top);          (* remember the base of the heap *)
  repeat
    root := nil;
    for i := 1 to 5000 do
      begin
        new(p);       (* after disposes, will allocate from free list *)
        p^.next := root;  root := p;  (* chain all cells together *)
        ...
      end;
    ...
    repeat            (* give back all cells one at a time *)
      p := root;
      root := root^.next;      (* follow the chain *)
      dispose(p);     (* mem manager puts on a free list *)
    until root = nil;
  until false;        (* program will run forever *)
end.
```

The recycling algorithm takes advantage of the fact that programs which use the heap operate on a great many variables of just a few types. Each type has a characteristic size. When a variable is disposed, it is saved at the front of a list of other variables of the same size. When a variable is allocated, the NEW routine first looks on the list corresponding to the size required; if there is a free object there, it can be allocated immediately. Usually there will be very little computational overhead for either NEW or DISPOSE.

The memory manager maintains free lists for objects of sizes 4, 6, 8, ... 32 bytes, and one more list for all larger objects. Objects are allocated from this last list on a first-fit basis. No dynamic variable is ever allocated an odd number of bytes.

It is possible for the program to behave so that the heap becomes fragmented (broken into many small pieces). If a request then arrives to allocate space for a large variable, the memory manager will try to recombine the fragments to make a piece big enough to satisfy the request. The fragments must be sorted by address and adjacent ones merged.

The recombination process takes much longer than a simple allocation. Consequently, in real-time applications it is important to analyze the dynamic behavior of programs which use DISPOSE.

Mixing DISPOSE and RELEASE

It is also possible to mix the regimes in a well-behaved manner. However, not all implementations of HP Pascal allow mixing these methods in a program. A program which does so may not run properly on other implementations.

If you RELEASE a properly MARKed pointer after some calls to DISPOSE, the memory manager will leave on the free lists all disposed objects whose addresses are below the released location. All the space above the released location becomes free, whether or not it was disposed.

During this process the memory manager also recombines any adjacent free fragments, so RELEASE can also be used to reduce fragmentation. Just MARK the current top of the heap, then immediately RELEASE to the same spot.

What Can Go Wrong?

This section discusses some problems which may occur when using the Compiler, and how to solve them.

Can't Run the Compiler

1. If the system reports, `Cannot open 'COMPILER'`, the volume with the Compiler is not online. You may have removed the volume and not put it back. If the Compiler wasn't found when the system booted, you are expected to insert the disc containing the Compiler in the drive before invoking it. The system is shipped with the Compiler on the diskette labelled CMPASM.
2. If the system reports, `Cannot load 'COMPILER'`, either the disc is bad or not enough memory is installed in the computer to run the Compiler. It is desirable to have at least 393 Kbytes; the system is normally sold with at least 512 Kbytes.

Errors 900 thru 908

During compilation, three files are written by the Compiler: the code file, which is the one you want, and the REF and DEF files. The latter two are temporary working storage for linkage information which is appended to the code file if the compilation terminates normally. All three of these files are normally opened on the same volume (the volume to which you directed the code file).

Each of these files is subject to three classes of error:

- Error in opening the file.
- Insufficient space to open the file.
- File fills up before compilation finishes.

An error in opening the file usually means the volume is not online. It can also indicate that the volume's directory is full.

The amount of space allocated to the code file is usually half of the largest free area on the volume, with the potential to expand to the second half of that area if needed. If you get errors 900, 903, or 906 you need to make more room on the volume to which the code file was directed, or use a different volume.

The REF file by default is opened with 30 blocks of disk space on the same volume as the code file. A Compiler directive at the beginning of the source program can change the size and the volume selected for REF. There's no simple rule which gives the "right" size for the REF file. If the file fills up (error 907), make it bigger in proportion to the amount of program that remained to compile when the error occurred.

`$REF 50$`	Allocate 50 blocks
`$REF 'V3:'$`	Put it on volume V3
`$REF 'V4:', REF 50$`	Put it on V4 and allocate 50 blocks

Exactly analogous remarks hold for the DEF file, except that its default size is 10 blocks and the directive is `$DEF$`.

Errors When Importing Library Modules

There are several errors that can occur when importing modules.

1. Syntax errors in the interface of an imported library module. This usually indicates that the library module itself tried to import some other module which was not found by the Compiler's search algorithm.
2. Errors 608, 610: Include or import nesting too deep. If module "A" imports "B", which imports "C" and so forth, the Compiler must follow the chain to its end. The chain can only be 10 imports deep (unless you use the `$SEARCH_SIZE$` option). Since the same file handling mechanism is used to process `$IMPORT$` and `$INCLUDE$` files, the combined limit on import and inclusion nesting is 10 deep (unless changed with the `$SEARCH_SIZE$` option.
3. Error 613: Imported module does not have interface text. If the library has been linked by the Librarian, the interface specification has been removed. Also, a main program looks internally like a module; but it has no interface text.

Not Enough Memory

If the Compiler generates error −2 "Not Enough Memory", there isn't enough room in memory to compile the program. You can watch the numbers which appear on the screen in square brackets as the compilation proceeds – they show approximately how much memory is left. There are two primary reasons for running out of memory during a compilation. One of them is large procedure bodies, and the other is permanently loaded ("P-loaded") files.

Large Procedure Bodies

When the Compiler processes a procedure, the entire procedure (declarations and body) is scanned. An internal representation of the procedure, called a "tree", is built. This tree is not complete until the scanner reaches the end of the procedure, and only then does code generation begin. The tree form takes a lot of storage, particularly the statements making up the body. If you write a procedure whose body is ten pages long, the Compiler is very likely to run out of memory. The moral is that you should keep your procedures reasonably short. A good guideline is that no procedure should be longer than a page or two.

P-loaded Files

If you've Permanent-loaded a lot of libraries or programs, or space has been allocated to a memory-resident mass storage volume, you can reboot the system to recover the memory, and try again.

Insufficient Space for Global Variables

You may discover, either at compile time or at run time, that there isn't sufficient space for the global variables of your program. If this happens, please refer to the *Implementation Restrictions* section in this appendix, which explains the limitations and what to do if you exceed them.

Errors 403 thru 409

These errors should never be reported by the operating system. They usually indicate a malfunction in the Compiler itself. (Although one may occur due to a strange coding error.) If this ever happens, please show the program which causes it to your HP field support contact.

Error Messages

This section contains all of the error messages and conditions that you are likely to encounter during the operation of your workstation.

- Run time errors – These may occur when you are running a program.
- I/O related errors – When run-time error −10 occurs, there has been a problem with the I/O system. The operating system then prints a message from the I/O error list.
- I/O LIBRARY errors – When run-time error −26 occurs, there has been a problem in an I/O LIBRARY procedure.
- Graphics LIBRARY errors – When run-time error −27 occurs, there has been a problem in a graphics LIBRARY procedure.
- Compiler syntax errors – During the compilation of a program, any of these errors may occur. The compiler will show the number of the error and you can look it up.

Unreported Errors

Certain errors in Pascal programs are not reported by this implementation.

- Disposing a pointer while in the scope of a WITH referencing the variable to which it points.
- Disposing a pointer while the variable it points to is being used as a VAR parameter.
- Disposing an uninitialized or NIL pointer.
- Disposing a pointer to a variant record using the wrong tagfield list.
- Assigment to a FOR-loop control variable while inside the loop.
- GOTO into a conditional or structured statement.
- Exiting a function before a result value has been assigned.
- Changing the tagfield of a dynamic variable to a value other than was specified in the call to NEW.
- Accessing a variant field when the tagfield indicates a different variant.
- Negative field width parameters in a WRITE statement.
- The underscore character "_" is allowed in identifiers. This is permitted in HP Pascal, but is not reported as an error when compiling with `$ANSI$` specified.
- Value range error is not always reported when an illegal value is assigned to a variable of type SET.

Operating System Run Time Error Messages

Errors detected by the operating system during the execution of a program generate one of the following error messages.

When using the TRY..RECOVER construct, the following numbers correspond to the value of `ESCAPECODE`.

0	Normal termination.
–1	Abnormal termination.
–2	Not enough memory.
–3	Reference to NIL pointer.
–4	Integer overflow.
–5	Divide by zero.
–6	Real math overflow. (The number was too large.)
–7	Real math underflow. (The number was too small.)
–8	Value range error.
–9	Case value range error.
–10	Non-zero IORESULT.
–11	CPU word access to odd address.
–12	CPU bus error.
–13	Illegal CPU instruction.
–14	CPU privilege violation.
–15	Bad argument - SIN/COS.
–16	Bad argument - Natural Log.
–17	Bad argument - SQRT. (Square root.)
–18	Bad argument - real/BCD conversion.
–19	Bad argument - BCD/real conversion.
–20	Stopped by user.
–21	Unassigned CPU trap.
–22	Reserved
–23	Reserved
–24	Macro Parameter not 0..9 or a..z
–25	Undefined Macro parameter.
–26	Error in I/O subsystem.
–27	Graphics error. RAM Parity error.
–29	Misc. floating-point hardware error.

IO Errors

These error messages are automatically printed by the system unless you have enclosed the statement in a TRY-RECOVER construct. Within a RECOVER block, when `ESCAPECODE` = −10, one of the following errors has occurred. You can determine which error if you examine the system variable `IORESULT`.

0	No I/O error reported.
1	Parity (CRC) incorrect.
2	Illegal unit number.
3	Illegal I/O request.
4	Device timeout.
5	Volume went off-line.
6	File lost in directory.
7	Bad file name.
8	No room on volume.
9	Volume not found.
10	File not found.
11	Duplicate directory entry.
12	File already open.
13	File not open.
14	Bad input format.
15	Disc block out of range.
16	Device absent or unaccessible.
17	Media initialization failed.
18	Media is write protected.
19	Unexpected interrupt.
20	Hardware/media failure.
21	Unrecognized error state.
22	DMA absent or unavailable.
23	File size not compatible with type.
24	File not opened for reading.
25	File not opened for writing.
26	File not opened for direct access.
27	No room in directory.
28	String subscript out of range.
29	Bad file close string parameter.
30	Attempt to read or write past end-of-file mark.
31	Media not initialized.
32	Block not found.
33	Device not ready or medium absent.
34	Media absent.
35	No directory on volume.
36	File type illegal or does not match request.
37	Parameter illegal or out of range.
38	File cannot be extended.
39	Undefined operation for file.
40	File not lockable.
41	File already locked.
42	File not locked.
43	Directory not empty.
44	Too many files open on device.
45	Access to file not allowed.
46	Invalid password.
47	File is not a directory.
48	Operation not allowed on directory.
49	Cannot create /WORKSTATIONS/TEMP_FILES.
50	Unrecognized SRM error.
51	Medium may have been changed.
52	IO result was 52.

I/O LIBRARY Errors

When run-time error −26 occurs, there has been a problem in an I/O LIBRARY procedure. The operating system puts a value in the system variable IOE_RESULT. By importing the IODECLARATIONS module, you can access IOE_RESULT and call the IOERROR_MESSAGE function, which returns the error description. For example:

```
$SYSPROG ON$
      ...
IMPORT iodeclarations
      ...
BEGIN
  TRY
        ...  {statements}
  RECOVER
      IF escapecode = ioescapecode
         THEN writeln (ioerror_message(ioe_result));
      escape(escapecode);
END.
```

ESCAPE is a procedure you can call and ESCAPECODE is a variable you can access when you use the $SYSPROG ON$ compiler directive. IOESCAPECODE is a constant (equal to −26) you can import from the IODECLARATIONS module.

1	No error.	**18**	Not system controller.
2	No card at select code.	**19**	Bad status or control.
3	Not active controller.	**20**	Bad set/clear/test operation.
4	Should be device address, not select code.	**21**	Interface card is dead.
5	No space left in buffer.	**22**	End/eod has occurred.
6	No data left in buffer.	**23**	Miscellaneous - value of parameter error.
7	Improper transfer attempted.	**306**	Data-Comm interface failure.
8	The select code is busy.	**313**	USART receive buffer overflow.
9	The buffer is busy.	**314**	Receive buffer overflow.
10	Improper transfer count.	**315**	Missing clock.
11	Bad timeout value.	**316**	CTS false too long.
12	No driver for this card.	**317**	Lost carrier disconnect.
13	No DMA.	**318**	No activity disconnect.
14	Word operations not allowed.	**319**	Connection not established.
15	Not addressed as talker.	**325**	Bad data bits/parity combination.
16	Not addressed as listener.	**326**	Bad status/control register.
17	A timeout has occurred.	**327**	Control value out of range.

Graphics Library Errors

When run-time error −27 occurs, there has been a problem in a graphics LIBRARY procedure.

By importing the DGL_LIB module and enclosing the main body in a TRY-RECOVER statement, you can call the GRAPHICSERROR function which returns an INTEGER value you can cross reference with the numbered list of graphics errors. For example:

```
$SYSPROG ON$
      ...
import DGL_LIB
      ...
BEGIN
TRY
      ...  {statements}
RECOVER
    IF escapecode = -27
        THEN writeln ('Graphics error #', graphicserror,
                      ' has occurred')
        ELSE escape(escapecode);
END.
```

You may wish to write a procedure which takes the INTEGER value from GRAPHICSERROR and prints the description of the error on the CRT. You could keep this procedure with your program or, for more global use, in SYSVOL:LIBRARY.

0	No error. {Since last call to graphicserror or init_graphics.}
1	The graphics system is not initialized.
2	The graphics display is not enabled.
3	The locator device is not enabled.
4	ECHO value requires a graphic display to be enabled.
5	The graphics system is already enabled.
6	Illegal aspect ratio specified.
7	Illegal parameters specified.
8	The parameters specified are outside the physical display limits.
9	The parameters specified are outside the limits of the window.
10	The logical locator and the logical display use the same physical device. {The logical locator limits cannot be redefined explicitly. They must correspond to the logical view surface limits.}
11	The parameters specified are outside the current virtual coordinate system boundary.
12	The escape function requested is not supported by the graphics display device.
13	The parameters specified are outside of the physical locator limits.

Pascal Compiler Errors

The following errors may occur during the compilation of a HP Pascal program.

ANSI/ISO Pascal Errors

1	Erroneous declaration of simple type
2	Expected an identifier
4	Expected a right parenthesis ")"
5	Expected a colon ":"
6	Symbol is not valid in this context
7	Error in parameter list
8	Expected the keyword OF
9	Expected a left parenthesis "(""
10	Erroneous type declaration
11	Expected a left bracket "["
12	Expected a right bracket "]"
13	Expected the keyword END
14	Expected a semicolon ";"
15	Expected an integer
16	Expected an equal sign "="
17	Expected the keyword BEGIN
18	Expected a digit following '.'
19	Error in field list of a record declaration
20	Expected a comma ","
21	Expected a period "."
22	Expected a range specification symbol ".."
23	Expected an end of comment delimiter
24	Expected a dollar sign "$".
50	Error in constant specification
51	Expected an assignment operator ":="
52	Expected the keyword THEN
53	Expected the keyword UNTIL
54	Expected the keyword DO
55	Expected the keyword TO or DOWNTO
56	Variable expected
58	Erroneous factor in expression
59	Erroneous symbol following a variable
98	Illegal character in source text
99	End of source text reached before end of program
100	End of program reached before end of source text
101	Identifier was already declared
102	Low bound > high bound in range of constants
103	Identifier is not of the appropriate class
104	Identifier was not declared
105	Non-numeric expressions cannot be signed
106	Expected a numeric constant here
107	Endpoint values of range must be compatible and ordinal
108	NIL may not be redeclared
110	Tagfield type in a variant record is not ordinal
111	Variant case label is not compatible with tagfield
113	Array dimension type is not ordinal
115	Set base type is not ordinal
117	An unsatisfied forward reference remains
121	Pass by value parameter cannot be type FILE
123	Type of function result is missing from declaration
125	Erroneous type of argument for built-in routine
126	Number of arguments different from number of formal parameters
127	Argument is not compatible with corresponding parameter
129	Operands in expression are not compatible
130	Second operand of IN is not a set
131	Only equality tests (=, <>) allowed on this type
132	Tests for strict inclusion (<, >) not allowed on sets

133	Relational comparison not allowed on this type
134	Operand(s) are not proper type for this operation
135	Expression does not evaluate to a boolean result
136	Set elements are not of ordinal type
137	Set elements are not compatible with set base type
138	Variable is not an ARRAY structure
139	Array index is not compatible with declared subscript
140	Variable is not a RECORD structure
141	Variable is not a pointer or FILE structure
143	FOR loop control variable is not of ordinal type
144	CASE selecter is not of ordinal type
145	Limit values not compatible with loop control variable
147	Case label is not compatible with selector
149	Array dimension is not bounded
150	Illegal to assign value to built-in function identifier
152	No field of that name in the pertinent record
154	Illegal argument to match pass by reference parameter
156	Case label has already been used
158	Structure is not a variant record
160	Previous declaration was not forward
163	Statement label not in range 0..9999
164	Target of nonlocal GOTO not in outermost compound statement
165	Statement label has already been used
166	Statement label was already declared
167	Statement label was not declared
168	Undefined statement label
169	Set base type is not bounded
171	Parameter list conflicts with forward declaration
177	Cannot assign value to function outside its body
181	Function must contain assignment to function result
182	Set element is not in range of set base type
183	File has illegal element type
184	File parameter must be of type TEXT
185	Undeclared external file or no file parameter
190	Attempt to use type identifier in its own declaration
300	Division by zero
301	Overflow in constant expression
302	Index expression out of bounds
303	Value out of range
304	Element expression out of range
400	Unable to open list file
401	File or volume not found
403..409	Compiler errors

Compiler options

600	Directive is not at beginning of the program
601	Indentation too large for $PAGEWIDTH
602	Directive not valid in executable code
604	Too many parameters to $SEARCH
605	Conditional compilation directives out of order
606	Feature not in Standard PASCAL flagged by $ANSI ON
607	Feature only allowed when $UCSD enabled
608	$INCLUDE exceeds maximum allowed depth of files
609	Cannot access this $INCLUDE file
610	$INCLUDE or IMPORT nesting too deep to IMPORT <module-name>
611	Error in accessing library file
612	Language extension not enabled
613	Imported module does not have interface text
614	LINENUM must be in the range 0..65535

620	Only first instance of routine may have $ALIAS
621	$ALIAS not in procedure or function header
646	Directive not allowed in EXPORT section
647	Illegal file name
648	Illegal operand in compiler directive
649	Unrecognized compiler directive

Implementation restrictions

651	Reference to a standard routine that is not implemented
652	Illegal assignment or CALL involving a standard procedure
653	Routine cannot be followed by CONST,TYPE,VAR, or MODULE
655	Record or array constructor not allowed in executable statement
657	Loop control variable must be local variable
658	Sets are restricted to the ordinal range 0 .. 255
659	Cannot blank pad literal to more than 255 characters
660	String constant cannot extend past text line
661	Integer constant exceeds the range implemented
662	Nesting level of identifier scopes exceeds maximum (20)
663	Nesting level of declared routines exceeds maximum (15)
665	CASE statement must contain a non-OTHERWISE clause
667	Routine was already declared forward
668	Forward routine may not be external
671	Procedure too long
672	Structure is too large to be allocated
673	File component size must be in range 1..32766
674	Field in record constructor improper or missing
675	Array element too large
676	Structured constant has been discarded (cf. $SAVE_CONST)
677	Constant overflow
678	Allowable string length is 1..255 characters
679	Range of case labels too large
680	Real constant has too many digits
681	Real number not allowed
682	Error in structured constant
683	More than 32767 bytes of data
684	Expression too complex
685	Variable in READ or WRITE list exceeds 32767 bytes
686	Field width parameter must be in range 0..255
687	Cannot IMPORT module name in its EXPORT section
688	Structured constant not allowed in FORWARD module
689	Module name may not exceed 15 characters
696	Array elements are not packed
697	Array lower bound is too large
698	File parameter required
699	32-bit arithmetic overflow

Non-ISO Language Features

701	Cannot dereference (^) variable of type anyptr
702	Cannot make an assignment to this type of variable
704	Illegal use of module name
705	Too many concrete modules
706	Concrete or external instance required
707	Variable is of type not allowed in variant records
708	Integer following # is greater than 255
709	Illegal character in a "sharp" string
710	Illegal item in EXPORT section
711	Expected the keyword IMPLEMENT
712	Expected the keyword RECOVER
714	Expected the keyword EXPORT
715	Expected the keyword MODULE

716	Structured constant has erroneous type
717	Illegal item in IMPORT section
718	CALL to other than a procedural variable
719	Module already implemented (duplicate concrete module)
720	Concrete module not allowed here
730	Structured constant component incompatible with corresponding type
731	Array constant has incorrect number of elements
732	Length specification required
733	Type identifier required
750	Error in constant expression
751	Function result type must be assignable
900	Insufficient space to open code file
901	Insufficient space to open ref file
902	Insufficient space to open def file
903	Error in opening code file
904	Error in opening ref file
905	Error in opening def file
906	Code file full
907	Ref file full
908	Def file full

HEWLETT PACKARD

Part No. 98615-90050
E 0284
Microfiche No. 98615-99050

Printed in U.S.A.
First Edition, February 1984